REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	12-21-95	Final 6-15-93 to 2-15-95

4. TITLE AND SUBTITLE

Next-Generation Automated Cognitive Skill Training: Problems, Goals, and Approaches

5. FUNDING NUMBERS

F49620-93-C-0030
61102F
2313/BS

6. AUTHOR(S)

John Leddo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Research Development Corporation
2875 Towerview Road, Suite A-4
Herndon, VA 22071

8. PERFORMING ORGANIZATION REPORT NUMBER

AFOSR-TR-96-0007

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Office of Scientific Research/NL
110 Duncan Avenue, Suite B115
Bolling AFB, DC 20332-8080
Dr John F Tangney

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release;
[illegible]

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Research efforts toward the next generation of training systems should have two goals. First, training should achieve significant improvements in speed and quality of skill acquisition. Second, development and production of computer-based training systems should advance beyond the current hand-crafting approach by devising both methodologies and tools that make it possible for a large number of developers to engineer many different high quality training systems. This paper focuses on two key problems that currently confront the computer-based cognitive skills training community: the needs for 1) strong models of cognitive skill acquisition that prescribe what students should learn and how they should learn it and 2) robust engineering methodologies that can support the rapid, widespread development of effective computer-based training for cognitive skills. (continued on reverse)

14. SUBJECT TERMS

intelligent tutoring systems, knowledge representation, problem solving, expertise

15. NUMBER OF PAGES

84 + appendices

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
U	U	U	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

19960129 139

DTIC QUALITY INSPECTED 1

29 DEC 1995

Next-Generation Automated Cognitive Skill Training:

Problems, Goals, and Approaches

Submitted by:

John Leddo

Research Development Corporation

2875 Towerview Road, Suite A-4

Herndon, Virginia 22071

The research contained in this report was sponsored by the Air Force Office of Scientific Research (AFMC), under Contract F49620-93-C-0030. The views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Air Force Office of Scientific Research or the U.S. Government.

1.0 Introduction

1.1 Statement of two problems

There are two problems that must be addressed if next-generation training systems for cognitive skills are to be 1) more effective than current systems and 2) developed and fielded on the scale that will be required to meet future training needs.

1.1.1 PROBLEM I: There Needs To Be A Theory That Prescribes How To 1) Teach Cognitive Skills That Require Multiple Forms of Knowledge & 2) Determine Whether The Student Is Acquiring Knowledge In The Correct Forms During Skill Acquisition.

If the purpose of cognitive skill training in the military is to produce expert performance (and it is), then training should develop in the learner the same knowledge and problem solving strategies that experts have. For example, the goal of training military personnel to solve algebra word problems, or to determine the causes of circuit faults, should be to teach them the knowledge and problem solving skills that are used by experts at solving algebra word problems or determining the causes of circuit faults. Indeed, at every step along the way to expertise, diagnosis and training should be directed toward that goal.

This line of reasoning has a significant implication for the next generation of automated training methods, if we want them to produce order-of-magnitude improvements over the current generation. Expertise at most interesting cognitive skills, such as solving algebra word problems, diagnosing circuit faults, requires that the expert use several forms of knowledge to obtain a correct answer (Leddo et al., 1988, 1990). For example, in the case of circuit fault diagnosis, at least three different forms of knowledge are required for expert performance:

o Knowledge Type 1: Objects and Attributes encode the properties of the entities in (faulty) circuits. Examples of objects are diodes, transistors, wires, breaks in wires, etc. Examples of attributes are impedance, thickness, etc.

o Knowledge Type 2: Mental Models that permit the problem solver to "run" the circuit mentally and produce inferences about the behavior of the circuit under different initial conditions i.e., attributes of objects in the circuit.

o Knowledge Type 3: Rules that encode diagnostic strategies, inferential rules for determining the cause of observed symptoms, repair techniques, and so forth.

In the case of algebra word problems as well, several forms of knowledge must be used to achieve a solution. The problem solver must have knowledge about the various algebraic forms of problem types -- time-distance problems; word problems; etc. -- rules that determine the operations to perform on the algebraic forms, and, of course, knowledge about the objects, attributes, and mental models that represent the domain elements referred to in the problem. Work performed at Research Development Corporation suggests that math practitioners both possess and use these diverse types of knowledge in solving practical problems involving algebra.

Thus, we believe that training most "interesting" cognitive skills will require teaching the learner several forms of knowledge. This implies that the next generation of computer-based trainers for cognitive skills will be required to respond explicitly to the requirement for teaching multiple forms of knowledge. Currently, however, most computer-based tutoring/training systems for cognitive skills use a single representation for the knowledge and skills they are trying to teach. Those that use multiple forms of knowledge do so for pragmatic reasons rather than because a strong cognitive theory of expertise and training prescribes them. For example, the representation for knowledge in tutoring systems based on

Anderson's ACT* theory is almost exclusively in the form of production rules and, where it varies from production rules, it is typically in service of the production rule representation. Similarly, work by Eliot Soloway and his colleagues (cf., Johnson and Soloway, 1985) focuses on frame-based representations of knowledge.

In short, if the expert knowledge required to solve target problems has a single form, then tutoring based on a single form of knowledge is likely to be effective. If, however, the expert knowledge required to solve target problems is not in a single form then, even though tutoring based on a single knowledge form may produce some positive effects, the tutoring may not be as effective or as efficient as it could be if it were not limited to a single form for the knowledge. In addition, students will be unlikely to achieve expertise without significant changes in the ways they represent the knowledge required for expertise.

The requirement for teaching multiple forms of knowledge imposes a further constraint on the assessment component of next-generation training systems. The further constraint is that the training system must, at every step in the training, monitor the forms in which the student acquires knowledge. For example, if expert knowledge about a device is represented in a mental model that encodes both structure and function, then the student should acquire it in that form and not, for example, in the form of a list of objects and attributes that represent components of the device. To accomplish this, the training system must have 1) explicit strategies for teaching the knowledge in the correct form and 2) assessing whether the student is indeed acquiring the correct form of knowledge.

If the student begins to diverge from the correct form for the knowledge the training system is trying to teach, then the tutorial component must correct the student. When there are many possible forms for some particular piece of knowledge, but only one form consistent with expertise, then the less the tutor knows about the form of the student's knowledge, the less effective it will be in rapidly getting the student back on track. The need for close monitoring arises because the tutorial planning required to get the student back on track may be quite complex (Littman, 1991) and, in some cases, impossible if the student gets too far afield.

In sum, we need 1) a theory that prescribes the forms of knowledge that should be taught by training systems for cognitive skills and 2) an on-line assessment methodology that will permit the training system to determine whether the student is acquiring the correct forms of knowledge.

The above analysis leads to two objectives to be achieved to progress to the next generation of intelligent training systems for cognitive skills:

Objectives: We need a theory that prescribes how to 1) use multiple knowledge forms to train specific cognitive skills and 2) determine whether the student is acquiring knowledge in the correct forms during skill acquisition.

1.1.2 PROBLEM II: There Is No Engineering Methodology To Support Rapid, Cost-Effective, Replicable Development & Maintenance Of Training Materials That Use Multiple Knowledge Forms.

Solving Problem I is only half the battle. Even if we devise a theory of multiple knowledge forms for cognitive skills that prescribes the content, structure, and tutoring methods for next-generation systems, there is no guarantee that it will be possible to produce the required quantity of effective, maintainable, and robust training systems for all the cognitive skills that we need to train. Currently, the development process for intelligent computer-based training is time-consuming and does not typically yield systems that are easily evaluated, maintained or modified. In addition, producing nearly any system is a start-from-scratch effort in which 1) little software built by others can be reused, 2) most software components produced for the system can only occasionally be reused in subsequent efforts within the development group and 3) software produced by one group is almost never reused outside the group. This state of affairs obviously limits the quality, consistency, and quantity of current training materials.

The task of constructing next-generation training for cognitive skills is bound to be more difficult. Future systems will be more complex, undoubtedly require the cooperation of many people to produce them, and will require significant maintenance and modification as delivery software, training requirements, and hardware change. In short, effective development and fielding of these systems will require a robust engineering methodology just as the design of large-scale software requires a software engineering methodology. Hence, the second critical problem that must be solved to progress to the next generation of intelligent training systems for cognitive skills:

Problem II: There is no engineering methodology to support rapid, cost-effective, replicable development and maintenance of training materials that use multiple knowledge forms.

1.2 Statement of two goals

In response to the two problems identified in the previous section, we have defined two goals for the proposed research and development effort. In the remainder of this section we state the two goals, justify them, and identify some of their implied subgoals.

1.2.1 GOAL I: Develop a Tutoring & Assessment Methodology For Cognitive Skill Training Based On Multiple, Integrated Knowledge Forms. Our case for using multiple forms of knowledge in intelligent computer-based training systems for cognitive skills consists of five main arguments. We feel that any one of the arguments would recommend the use of multiple knowledge forms. However, the five arguments touch on most essential aspects of the design, development, and operation of training systems. Thus, we believe that a strong case can be made for developing the approach of multiple forms of knowledge. In the remainder of this section we identify and briefly describe the arguments. The arguments are quite intuitive and most were touched on in the introductory section of the report.

Argument 1: Cognitive Reality of Multiple Knowledge Forms. It is clear, from intuition and from the cognitive science literature on instruction and expertise, that many problem solving tasks require multiple forms of knowledge. If the goal of cognitive skill training is to foster expertise, then students should learn what experts know. This implies that, in many instances, training systems will have to teach students knowledge in several forms.

Argument 2: Accuracy of Diagnosis and Assessment of Learning. If a training system cannot reason about different forms of knowledge, then it cannot have a robust assessment methodology for determining whether, during the training process, the student is acquiring knowledge in the form appropriate for expertise. If a training system's only measures of performance are which problems the student correctly solved and which the student incorrectly solved, and assessment problems are not generated explicitly to determine the form in which the student has acquired knowledge, then the system will not know why the student correctly solved some problems and incorrectly solved others. For example, a student may give the same wrong answer because he made a careless mistake (but generally understands the material), misremembers a particular fact (but generally understands the process he's using), or is thoroughly confused and is largely guessing. Each of these has different implications for corrective instruction. Perhaps an even more dangerous case is when the student achieves a correct answer by mimicking a procedure without understanding why the procedure works. Here the student may get the textbook problem "right" but fail miserably when placed in a real-world context and given a somewhat novel task that requires modification of the procedure.

If one of the criteria of achieving expertise is that knowledge be in the appropriate form, then unless the training system can assess multiple knowledge forms and which of those forms the student is using, it will not be able to assess expertise effectively. Thus, multiple knowledge forms are necessary to the assessment function of training systems for many cognitive skills.

Argument 3: Efficacy of Tutoring. If the knowledge that a training system is intended to teach can only have one form, then there is no ambiguity about how the student should represent it. For example, if a training system is teaching only diagnostic rules, then the likelihood that the student will not represent the knowledge as rules is negligible and tutoring will consist primarily of refining the condition and actions parts of the rules. If, however, there are several plausible forms for some piece of knowledge then there can be ambiguity about 1) what form the student's knowledge is currently in and 2) which tutoring strategies are appropriate to correct the student's understanding. For example, there may be a choice about whether to represent the parts of a device as a list of objects and attributes or as a coherent set of objects with behavior -- i.e., a mental model.

Unless the tutor knows what the alternative knowledge forms are, can assess which form the student's knowledge is in, and can reason about how to help the student change from one form to another, the tutor will be unlikely to help the student acquire knowledge in the forms appropriate for acquisition of expertise. Indeed, the student may need to learn the same knowledge in several forms to achieve expertise.

Argument 4: Economy of Storage and Computation. It has been argued that knowledge encoded using any of the popularly proposed knowledge representations can be transformed to any other representation. Nevertheless, it is clear that some representations are more naturally suited to certain representations than others. This can be for reasons of storage economy or computational efficiency. For example, a script (cf., Schank, 1982; Schank and Abelson, 1977) could be encoded as a sequence of production rules, each set to fire after the previous rule executes. The special structure of "script-like" rules can be exploited to economize storage either by not duplicating elements common to all scripts, or by storing features relevant to the whole script only once and not with each rule. In addition, special purpose, efficient inference engines can be built to take advantage of the script structure. For example, the entire script can be run in sequence, rather than requiring the pattern matcher to check the preconditions for all rules in the rulebase after each rule is fired).

Of course, these were some of the arguments that led researchers to postulate and implement the different representations now in current use. Cognitive scientists postulate similar explanations for experimental findings pointing to multiple representations in humans. Thus, it seems reasonable to carry the development of the theory of multiple representations to cognitive skill training.

Argument 5: Expressiveness for Designers of Training Systems for Cognitive Skills. Developers of training systems for cognitive skills need an intuitive language in which to express the knowledge required for expertise. For example, if a development environment provides a designer of a training system for circuit diagnosis with constructs that allow the designer to directly express the mental model knowledge, the diagnostic rule knowledge, and the object and attribute knowledge, then it is likely that the resulting system will 1) initially contain more knowledge appropriate for training expertise, and 2) be easier to change and augment than if it were built with a single form of knowledge. Equally, a developer of an algebra tutor who can directly express algebraic forms and manipulation rules will be likely to produce an initial prototype that is robust and cognitively effective.

1.2.2 GOAL II: Develop An Intelligent Reusability Engineering Methodology For Training Materials Development Process.

The design and development of training materials for cognitive skills is a complex, costly, and often inefficient process that requires the coordinated efforts of a diverse group of individuals. Problems commonly associated with this process include: 1) duplication of previous design and development efforts due to lack of communication across development groups and lack of a "corporate memory;" 2) miscommunication among group members due to differences in approach, background, terminology, etc.; 3) inappropriate products because of a failure to involve end users in crucial design decisions; and 4) suboptimal products resulting from imperfect group collaboration.

To improve the efficiency of the production and maintenance of training materials, personnel who construct them should have intelligent computer support for reuse of existing materials. In this engineering scenario, the developer could specify characteristics of training materials under construction and the intelligent support system would either 1) find appropriate reusable components in its knowledge base or 2) assist the user in constructing new materials which could then be added to the knowledge base. Until such support exists, production and modification of training materials is likely to be costly, prone to error, and subject to the "reinventing the wheel" syndrome.

We have, therefore, defined two primary subgoals, which we now identify and describe.

Subgoal 1: Develop a knowledge-based environment for constructing training materials that utilize multiple forms of knowledge. Accurately representing domain knowledge in a training or tutoring system is extremely difficult. Many times, developers of training system have the intuition that the expertise they want to train requires multiple forms of knowledge. To accommodate this intuition, they essentially wind up building special purpose languages that allow them to express domain knowledge in the forms required for expertise. We have the goal of building a general purpose, transportable knowledge based environment that would help developers of training systems 1) select appropriate forms for domain knowledge and 2) represent the knowledge. In addition, we intend to develop at the same time a mechanism that would store, for reuse, all the knowledge that individual developers represent. This would give anyone building training systems access an initial library of reusable domain knowledge to which they could contribute. We intend to develop a framework for the reusable knowledge and to provide software tools to help developers integrate it into their own training software. Ultimately, we would like to distribute this knowledge base on a hard disk as part of a knowledge based development package for training systems.

The knowledge-based environment will initially be constructed to support the development of training materials that use the five forms of knowledge identified in the introduction. In the first implementation, we focused on helping developers of training systems to represent domain knowledge using these forms of knowledge. Future work would 1) provide support for additional forms for domain knowledge and 2) extend the system to be able to support the representation of the forms of knowledge required for tutoring (Littman, 1989). This extension of the system would provide knowledge-based support for constructing tutors and trainers that use strategies appropriate for their domains of instruction. Again, as in the case of domain knowledge, we intend to build a mechanism to permit storage and reuse of tutorial knowledge.

Subgoal 2: Begin to close the Design-Production-Evaluation-Modification (D-PE-M) Loop. Training material often needs to be modified. Reasons for modification include changes in the work environment, technology and equipment, the structure of tasks, and the methodologies and results of evaluation.

For example, recent developments in cognitive theory produce well-defined assessment methodologies for cognitive skill acquisition that could be integrated into training programs. The effectiveness of training systems could thus be constantly assessed. Modifying the system for optimal performance could thus become a frequent requirement. Such modification would probably not be consistent if it were done by hand. At the very least, version-control methodologies from software engineering should be used to enforce the consistency of such modifications.

Other, larger scale, modifications would also require close adherence to established software engineering methods. For example, when new training technologies, such as virtual reality, become widely available, it would be advantageous to be able to introduce them into existing training software without ruining it. There are thus many reasons for evaluating and modifying training software and we believe that the knowledge based support environment should accommodate as many as possible. We therefore focused on producing a knowledge based support environment for designing, producing, evaluating, and modifying training software.

2.0 Background

Today's high-tech world requires a higher level of technical skill and literacy than ever before. The Air Force workplace increasingly requires its personnel to have technical skills. Unfortunately, recent educational assessments (cf., National Assessment of Educational Progress, 1989) show that graduating high school seniors are severely lacking in technical skills. As a result, employers such as the United States Air Force must bear an increasing burden in training its personnel.

One area that has shown immense promise is the use of automated trainers. Such trainers provide the student with one-on-one training, which has often been viewed as the most effective in terms of speed of learning (cf. Bloom, 1984). Similarly, such trainers offer hope as a means of reaching large numbers of students and in many cases are more economical than standard classroom instruction.

In the field of automated trainers, the field of intelligent tutoring systems (ITSs) has received considerable attention (cf., Brna, Ohlsson and Pain, 1993; Greer, 1995). Intelligent tutoring systems typically are driven by a cognitive model of the skills the student is trying to learn and have a cognitively-based instructional framework for how those skills can be developed.

Why use ITSs in training? There is clear evidence documenting the superiority of individualized instruction over traditional group instruction (e.g., Bloom, 1984). The superiority of small class sizes is usually attributed to the increased opportunity for individualized attention. The basic premise of the mastery learning paradigm (Bloom, 1968) is that the amount of instructional time allotted to a given objective should be allowed to vary among students, so that each student has the opportunity to master the objective before moving on to other objectives. Bloom (1984) showed that mastery learning improved performance by 1 standard deviation; adding individualized instruction improved performance by 2 standard deviations.

A major benefit of computer-aided instruction is the ability to tailor instruction to the individual student. An ITS differs from a traditional computer-aided instruction system in that it has an embedded model of the student's current understanding of the subject matter, and tailors instruction according to this model. ITS development has been an active area of research in the cognitive science and AI fields (Brna, Ohlsson and Pain, 1993; Polson and Richardson, 1988).

Very many ITSs have been developed over the last ten years, but only a few have been subjected to formal evaluation. Of course, the ones for which evaluations are reported constitutes a biased sample; but bearing this in mind, the results of the evaluations are quite impressive (Shute and Regian, 1990). For example, students having 20 hours of instruction from Sherlock, which teaches avionics troubleshooting (Lesgold, Lajoie, Bunzo and Eggan, 1990), performed on the evaluation test comparably to technicians with almost 4 years experience. Students using the LISP tutor (Anderson, Farrell and Sauers, 1984) learned the computer language LISP in half the time it took traditional classroom students.

3.0 Theoretical Framework.

3.1 The testbed.

The first task of the project was to select a testbed. The purpose of the testbed was twofold. First, it provided a basis for testing an instructional methodology based on our Integrated Knowledge Structure (INKS) framework. Second, the testbed served as a basis for developing a generic intelligent tutoring system architecture.

The INKS framework itself deals with conceptual (e.g., as compared to motor or perceptual) problem solving tasks. Therefore, selection of a testbed was driven by finding a problem solving domain that was largely conceptual and lent itself well to teaching each of the different types of knowledge in our INKS framework. We also wanted to select a domain with a wide range of applicability to real world problem solving. For this reason, scientific inquiry was chosen as the testbed for testing the INKS-based instructional approach.

This testbed was later changed to algebra for purposes of building the generic ITS architecture. This was motivated by the fact that an algebra ITS was under development at Armstrong Laboratories at Brooks AFB in San Antonio, Texas. Researchers were looking to test alternative approaches to ITSs, so the project team changed the testbed to accommodate this interest within the Air Force.

3.2. The instructional approach.

The first step in the development of an instructional approach is to develop a model of the testbed knowledge. Since our goal is not simply to present a curriculum within the context of a simulation but rather to build the same type of thinking skills in students that experts have, it is important to identify and model those skills. This model will then drive instructional requirements designed to build similar knowledge in students as well as drive the evaluation of how well students have learned that knowledge. These instructional requirements and evaluation feedback then drive what simulations/events the students experience (see below).

3.2.1 Expertise: Characteristics and rationale as a training target.

Given that we have set a goal of teaching students to solve problems as experts do, we want to develop a framework for modeling expert problem solving knowledge. There are several reasons why understanding how experts solve problems and represent knowledge has relevance to the proposed work. First, the goal of training is to make students effective problem solvers in the topic area taught. Research by Wagner and Sternberg (1985) shows that experts are effective, not just intelligent.

Second, research indicates that expert knowledge is diverse and well integrated (Laskey, Leddo, and Bresnick, 1989; Leddo et al., 1990). Experts have a variety of problem solving strategies at their disposal and can apply them as called for by the situation. These strategies are functional in nature and are oriented toward the goals and objectives that characterize the expert's job. As a result, the expert problem solving strategies are well integrated with everyday knowledge and are readily retrievable and applicable.

Third, experts tend to show a deep understanding of their subject area. Non-experts tend to be more superficial in their understanding and this can affect problem solving. For example, in physics, Chi et al. (1981) found that non-experts judge the similarity of problems on the basis of superficial features such as type of apparatus, while experts judge similarity by reference to basic principles of physics (e.g., conservation of energy) and generic solution techniques associated with such principles. Similar differences between experts and novices in algebra are reported by Schoenfeld and Herrmann (1982) and in computer programming by Weiser and Shertz (1983).

A central theme in expert knowledge is its functional orientation. Expert knowledge is centered around goals. As a result, any modeling of expert knowledge and implications derived for instruction must take goals into account. In fact, research by the Yale University Cognitive Science Group (cf., Galambos, Abelson and Black, 1986) suggests that goals play a powerful role in organizing people's knowledge in general. This point is important because much traditional classroom-style instruction is done without much emphasis on goals. Rather the focus is on problem solving procedures. While researchers such as Anderson (1982) have argued that expert knowledge is characterized by procedural knowledge, Leddo et al. (1990) find that true expertise is characterized by goal and causal knowledge while procedural knowledge actually characterizes experienced non-experts (who are more advanced than novices but are not true experts).

3.2.1.1 Modeling expert knowledge. The research described above focuses on the behaviors exhibited by experts. To translate knowledge of these behaviors into practical instruction and evaluation techniques, one must understand the types of knowledge that underlie these skills.

In the cognitive science and psychology literatures, several frameworks have been proposed as models of expert (and non-expert) knowledge. These schemes tend to address different types of knowledge. For example, scripts (Schank, 1982; Schank and Abelson, 1977) are used to represent goal and planning knowledge that is used in fairly routinized environments. Scripts are generalized sequences of steps used to achieve a goal. Script-like schemas can also be used to integrate bodies of knowledge into a larger framework.

Knowledge about data patterns and how objects are organized together can be represented by object frames (c.f., Anderson, 1980; Minsky, 1975). Frames are very much like scripts in that they are expectancy-driven organizers of knowledge. We conceptualize scripts as focusing more on goal and plan-related knowledge while frames organize collections of objects. Frames can also be distinguished from semantic nets (cf., Quillian, 1966) which tend to organize information about individual concepts and relationships between them rather than collections of objects. For example, a science laboratory may best be represented by a frame since it is a collection of people and equipment while a test tube may best be represented by a semantic net that describes its features.

Knowledge about situation-specific procedures can be represented by production rules (cf. Newell and Simon, 1972). Production rules are expressed in the form "IF [antecedent], THEN [consequent]", where antecedents are situational conditions that determine when procedures are to be executed and consequents are the procedures executed under those conditions. Production rules are useful in both carrying out procedures (e.g., "If this step has been completed, then do this next step.") and also generating inferences (e.g., "If the following problem features are observed, then infer that this is an [X] type of problem."). Production rules can be distinguished from scripts in that scripts organize entire goal-driven plans, while production rules organize specific actions. Scripts can be viewed as collections of production rules much the way that frames can be viewed as collections of semantic nets.

Finally, causal and analogical reasoning can be captured by mental models (cf., de Kleer and Brown, 1981; Johnson-Laird, 1983; Leddo, Cardie and Abelson, 1987). In our framework, (Leddo, Cardie and Abelson, 1987), mental models are viewed as encoding the causal rationale for why a specific problem solving procedure is used. One of the factors that distinguishes the way experts solve problems from the way non-experts do is the former's heavy reliance on mental models and the ability to use them to select an appropriate problem solving strategy to meet a set of objectives.

We have discussed five different representation frameworks (scripts, object frames, semantic nets, production rules and mental models) for representing expert knowledge. As we mentioned above, experts possess diverse knowledge that is richer that can be handled by any single framework. Leddo, Cardie and Abelson (1987) developed an Integrated Knowledge Structure (INKS) framework that combines these individual schemes. In the INKS framework, scripts serve as the general organizer of knowledge, linking plans and goals together. Production rules give situation-specific procedures to be executed given conditions that arise during the execution of a plan. Frames organize collections of objects that are utilized in the execution of plans while semantic nets organize features of the individual objects within a frame. Mental models provide the rationale for why procedures are executed and how they are instrumental in achieving objectives.

3.2.2 The development of expertise.

The INKS model may serve as a useful explanatory mechanism as to how experts acquire practical problem solving skills. John Anderson (1982) argues that expertise is acquired in stages. Initially, Anderson argues, people have declarative knowledge. Declarative knowledge is largely semantic. Anderson terms such knowledge as "knowing that." For example, a student learning to drive may know "that" he needs to put the key in the ignition and turn it while simultaneously pushing down on the gas peddle. The student "knows that" this is what he needs to do, although he still may not be capable of performing the act.

The next stage in acquiring knowledge is to proceduralize the declarative knowledge. This knowledge of what one is supposed to do is translated into concrete procedures that lead from problem to solution. Anderson terms this knowledge "knowing how" in that the student now knows how to start the car, shift gears, etc. This type of knowledge is represented by production rules. In essence, the student has transitioned from a more abstract representation of what he must do ("knowing that") to a more specific representation ("knowing how") that allows him to do it.

The drawback with Anderson's framework is that it forces a person to learn a specific set of procedures for every type of problem rather than generating a general formula from which different procedures might be generated. Leddo et al. (1990) extended Anderson's framework by studying the problem solving processes of people at different levels of expertise.

Leddo et al. found support for Anderson's two stages of skill development. However, the most experienced group of problem solvers, the true experts, used a different problem solving strategy. Experts used their goals and a causal understanding of the problem domain (i.e., a mental model) to select and adapt a strategy to fit the specific problem. Hence, the experts' problem solving process was more powerful than a procedural one, because experts had the means to generate procedures as needed from richer, more abstract knowledge structures. These procedures were causally related to both the situation and the goal. Leddo et al. termed this type of reasoning "knowing why."

We believe that this "knowing why" is what distinguishes experts from non-experts. We also feel the causal links between the goals and the planning strategy used with the specific procedures and semantic features of the situation are the keys to effective practical problem solving. (This is why all of the knowledge structures in the INKS framework are relevant and no single structure is sufficient.) We further argue that most students never advance beyond the "knowing how" stage (in fact, Leddo et al., 1990, found that only about 5% of the professional problem solvers they studied, all of whom had more than 20 years of professional experience, could be characterized as "knowing why" problem solvers). Students may have a fundamental understanding of how to apply a problem solving procedure, but lack a deeper understanding of why that procedure is relevant and what the purpose of that procedure is. As a result, unless they are given a specific procedure to follow in a practical problem solving setting, they will have great difficulty coming up with one on their own. On the other hand, if they can succeed in learning why problem solving procedures are relevant, we hypothesize that they will be better able to link these problem solving procedures to concrete, practical problems. (Shavelson, Webb, Stasz and McArthur, 1987, echo our sentiments by arguing that math education should include goals and underlying causal reasoning behind the subject matter).

Research at RDC (Leddo, Campbell, Black, and Isaacs, 1992) provides preliminary support for the contention that solving problems "procedurally" is not as effective as solving problems "causally." Algebra students when confronted with a practical problem typically look for a formula to execute. We believe they are performing problem solving procedurally--trying to map formulas (procedures) to the problem. This causes problems when the chosen formula does not fit the problem (e.g., the formula is geared toward solving an equation with one unknown but the problem has two unknowns). Math practitioners on the other hand who solve the same problem by establishing goals and developing a model of the problem (i.e., using the "expert" problem solving approach described above) have no difficulty with the same problems. Further, when the students who are having difficulty with the problems are asked to "forget about the formulas and develop a model of the problem," they often are then able to solve the problem.

3.2.3 Implications for Instruction.

We argue that our conceptualization of the knowledge skills underlying expert practical problem solving has important implications for how training can be conducted. We believe that instruction should focus on building the different types of knowledge outlined earlier, with emphasis on integrating them through causal mental models. Successful problem solving requires two major steps, both of which involve the use of integrated knowledge. First, students must analyze a problem, using the problem features presented, and infer an appropriate problem solving strategy. We view this as a "bottom-up" process in that students work from specific problem features to generate a more abstract problem representation.

Second, once a student has structured the problem and selected a strategy or set of problem solving procedures, s/he must then apply that strategy to the specific problem. This often involves mapping features of the problem onto slots in the strategy. We view this as a "top-down" process in that students work from the general strategy in order to instantiate it into a concrete problem.

Research by RDC team members (Leddo et al., 1992) supports this contention. We have found that math practitioners (those that use math as part of their professional jobs) do not reason by formulas when solving practical algebra problems. Rather, they use semantic and causal knowledge of the problem situation to weed out relevant from irrelevant information and then establish relationships among the different pieces of information. This is used to establish a problem solving strategy. (We view this process as a bottom-up structuring of the problem). Once the problem solving strategy is selected, the practitioners pick specific formulas and execute them. (We view this process as top-down problem solving). Interestingly, the formulas selected are often not the formulas that are "prescribed" by algebra, e.g., y=mx+b, but work just the same. This process is quite different than the formula-driven reasoning typically taught in algebra (and other) courses. However, our results suggest that math practitioners don't reason "mathematically" (i.e., by formulas), but rather use real world knowledge to structure the problem and then used math to solve it. We believe the biggest training challenge lies in teaching people to integrate their real-world reasoning skills with domain-specific problem solving skills and that this may in fact produce the biggest gain in learning.

Both of the two steps described above involve integrating abstract, formula-type knowledge with concrete, everyday knowledge. Students often have trouble with both steps. They may fail to recognize what "type" of problem it is from its description and/or they may pick a strategy but then not know which values presented in the problem go in which slots in the strategy. We hypothesize that causal mental models can help the students make these linkages. We outline a training approach to accomplish this.

First, the student is taught in the context of specific goals. The student is asked to solve a problem where no quantitative analysis is performed. Here, students may be presented with simple questions to investigate such as "does putting candy on sale lead to increased sales?" Students may think about what general considerations and issues need to be addressed to answer such a question. The goal is to induce students to build a model of the problem before they can solve it, without simply jumping into a set of procedures. In this case, the ITS would present the student with a variety of problems that depict different scenarios that the student might encounter. This gives the student a breadth of experience and makes salient the fact that there is a wide range of problems the student may have to encounter in real-world problem solving (one tendency novices have is to underestimate the variety of problem types and as a result often employs "one size fits all" problem solving).

Next, students are introduced to the framework for solving problems. For example, in the scientific inquiry context text, students would be taught the concepts such as what a hypothesis is, how to test a hypothesis in real world settings, etc. This stage can be viewed as Anderson's "knowing that" as students are learning the framework. Here, the ITS might show students scenarios with specific cues highlighted (e.g., sales at two stores, one having a sale, the other not) so that the student can see the relationship between concepts to be learned and their real world occurrences.

Next, students are given practice with this framework so that it can be proceduralized ("knowing how"). Here, the ITS would generate specific problem instances and the student must perform the procedures that s/he has been taught such as analyzing the data collected in the problem.

Finally, students are taught to develop an overall causal understanding or model of what they have learned (much as experts have). This involves giving students multiple problems with themes. An example of this in the scientific inquiry context might be to have the ITS simulate different experiments in which the samples on which data are collected are biased in some way so that the student can use this larger context to anticipate the specific affects that such biases may have on the validity of generalizing experimental results to an entire population. We hypothesize that this type of training is especially effective in building problem solving skills in time stressed environments as students learn to use context knowledge to make more rapid conclusions (i.e., the difference between "top-down" and "bottom-up" problem solving).

3.2.4 Diagnosing what students have learned using knowledge elicitation.

Knowledge elicitation refers to the process of ascertaining what a person knows about a given topic area and how that knowledge is organized. The goal of knowledge elicitation is to transform information obtained from a person into some representation of the person's knowledge.

There are two common categories of knowledge elicitation techniques. The first is called protocol analysis (cf. Ericsson and Simon, 1984) in which people articulate their thought processes while solving problems. Protocol analysis is good for eliciting what people know, in particular their practical problem solving procedures, but has serious limitations for eliciting their underlying organization and understanding of that knowledge.

Protocol analysis is also extremely tedious and time-consuming. In an intelligent tutoring application, standard protocol analysis is clearly infeasible. However, one can build into the ITS data-collecting procedures for monitoring students' problem-solving, and one can use querying to gather information about what students are thinking as they solve the problem.

The second type of commonly used knowledge elicitation technique is an interview or question and answer format. Leddo and Cohen (1989) report an interview technique called Cognitive Structure Analysis (CSA) that is explicitly designed to elicit people's underlying organization and understanding of their problem solving processes. CSA is based on the INKS framework described earlier. It involves questions regarding people's problem solving goals, the strategies they use (what procedures, what sequence they occur in, etc.), the reasons behind these strategies, the features of the problem that are relevant, etc.

CSA has received a preliminary testing in ITS implementation in THINKER (Leddo, Sak and Laskey, 1989). CSA was used to elicit from experts the domain knowledge that went into forming THINKER's knowledge base. This knowledge base formed the basis for THINKER's curriculum. THINKER also uses CSA to probe students when they make errors. The goal of these probes is to infer students' underlying knowledge gaps or errors in reasoning. THINKER compares what the student knows to its expert knowledge base to make this inference which then determines what corrective instruction THINKER gives the student. Separately, CSA was implemented as an automated assessment tool to assess the effectiveness of classroom instruction. The results of this study are discussed later in the report. Finally, CSA was used in the ITS architecture developed under the present project.

CSA's probes are driven by both structural and content considerations of the knowledge structures in the INKS framework. For example, scripts not only describe what events will happen in a context (the content knowledge), but also their sequence, their importance, etc. (the structural knowledge).

These content and structural considerations not only help capture what a student knows but also help distinguish the representation the student is using. For example, a production rule representation may capture event sequences, but is typically not used to capture relative event importance. Therefore, if a student is thinking in terms of event importance, s/he is probably using a script instead of a production rule representation.

Leddo et al. (1992) developed a hybrid technique that uses both problem solving and CSA techniques. In this technique, a student first solves a problem using his normal problem solving strategy. The steps he goes through are then recorded. Next, the elicitor revisits each step using the CSA technique in order to diagnose the knowledge used in that step. This technique is useful in generating a global model of problem solving knowledge as opposed to a highly detailed model that might be used to construct an expert system. The goal in our application is to give someone (student or tutor) a quick means to assess the student's level of reasoning. It is therefore important to be able to assess a global model, and much less important to assess the details. The other benefit of the hybrid technique is that it uses both natural problem solving and interview-based techniques. Some students may have difficulty articulating their knowledge. This makes it difficult to infer whether the problem is lack of understanding or inability to articulate their understanding. The hybrid technique allows students to demonstrate proficiency behaviorally as well as explaining it. This helps to distinguish which of these two factors is at work.

This hybrid knowledge elicitation technique is especially useful in simulation-based ITS environments. Students' interactions with the simulations can serve as a rich source of information about the students' naturally problem solving approaches. Further, scenarios can be manipulated by the ITS to see how these problem solving strategies differ, if at all, across situations.

This type of assessment, namely recording the students' problem solving process, is the means by which most ITSs construct a student model. We believe that this approach alone is insufficient as it does not give underlying reasoning about why the student took whatever actions he or she did. We believe understanding the student's underlying reasoning is key to developing corrective instruction. For this reason, we see the addition of CSA to the assessment as a valuable supplement.

In the present project, the goal was to use knowledge elicitation to identify discrepancies between student and expert. These discrepancies generate conclusions about the student's level of development which then guide the student's progress. For example, we argued above that novices are largely pattern-oriented (with simple patterns), experienced people are largely procedural and experts are goal and mental model-driven. If we are to help students learn to be goal and mental model-driven, we need to be able to determine what drives their current problem solving. This involves not only comparing students and experts in terms of the knowledge they have, but also comparing the student's knowledge with her behavior. In other words, if a student has a causal model but still reasons procedurally, she is not behaving as an expert would and therefore likely needs stronger linkage between her abstract causal knowledge and her concrete procedural knowledge. This is why CSA is especially important--it gets at the problem solving knowledge that underlies problem solving behavior. Once the student's cognitive level is assessed, a learning strategy can be developed based on the framework described in section 3.2.3.

4.0 An experimental test of the INKS-based instructional framework.

4.1 Overview

The first goal of the project was to develop an instructional framework suitable for intelligent tutoring systems that has the goal of teaching students how to solve problems as experts do. As discussed earlier, a fundamental premise of the present research is that expert problem solving knowledge is diverse and is best represented by a rich, integrated structure such as the INKS framework discussed earlier.

The Research Development Corporation methodology for developing such an instructional framework is comprised of the following steps:

1) Conduct knowledge elicitation with domain experts to identify the knowledge and skills that need to be taught.

2) Model this expert knowledge using INKS to understand the relationship between the knowledge concepts and how they are used in problem solving.

3) Develop instructional activities designed to teach these knowledge and skills.

4) Conduct knowledge elicitation with students to determine what knowledge is needed.

5) Select instructional activities designed to teach the needed knowledge.

6) Iterate steps 4) and 5) (and others as needed)

In order to leverage the current effort, steps 1) and 2) were completed by using work from a separate project. The subject area was scientific inquiry using statistical data analysis. Figure 1 presents an INKS representation of part of the knowledge elicited in this subject area.

4.2 <u>Method</u>

Subjects.

Two schools participated in the study. One was the New High School, an alternative school in Boulder, Colorado that served an at-risk student population. Eighteen students (16 9th graders and 2 10th graders) were selected from the New High School to participate in the experimental condition. The second participating high school was Boulder Valley High School, a mainstream high school, also located in Boulder, Colorado. From this high school, two control groups were constructed: a group of eleven 9th graders and a group of nine 12th graders.

An INKS Model of Statistics/ Scientific Inquiry

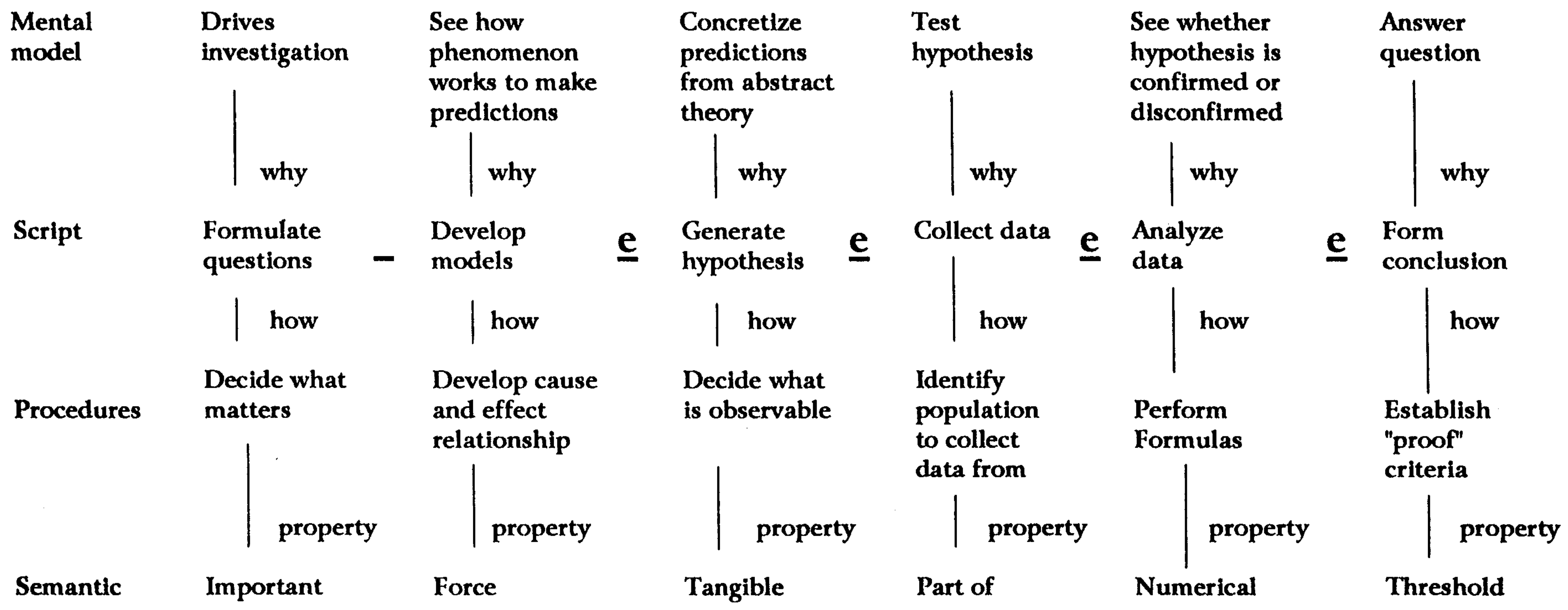


Figure 1

Materials.

A variety of performance and other problem solving tasks were constructed to teach the scientific problem solving method. These ranged from complete studies that emphasized the entire scientific method (as discussed earlier) to component tasks that emphasized specific problem solving skills. The emphasis of all problem solving tasks was to present real world problems that the students could relate to. An example of a task that emphasized the complete problem solving process was having the students construct, carryout and evaluate a study that addresses the question of whether putting an item on sale actually increases sales. An example of a task that emphasizes specific component skills was to present students with salary information for computer scientists and have them determine whether there was a bias due to gender once employees' experience and education was taken into consideration.

All told, 25 hours worth of instructional materials were developed. These were accompanied by relevant manipulatives (e.g., candy for the experiment to test the effects on sales of putting candy on sale, decks of cards for the study on ESP).

Procedure.

All subjects were paid volunteers who were recruited with the help of their schools. Subjects in the experimental condition were paid $80 for their participation in the project. Subjects in the control condition were paid $20. Prior to the start of the study, all students were given a pre-test on their knowledge of the scientific problem solving process. Questions were based on the INKS model of scientific inquiry as shown in Figure 1. Questions were multiple choice and comprised of knowledge-based questions and applied questions. An example of a knowledge-based question was "In scientific inquiry, defining the question you are asking is important because". An example of an applied problem was to calculate the mean of a set of test scores. All questions were followed by five possible answers. The pre-test and post-test were administered and scored via computer.

The experimental students were given 25 hours instruction over a period of three weeks. These included 1.5 hour sessions after school during the week and 4 hour sessions on Saturday. Given that students were expected to have after school activities, they were allowed to pick three afternoons a week for instruction. All students were required to attend on Saturday. Activities were done either individually, in small groups or as an entire class. Three RDC staff members participated as instructors.

Upon completion of the 25 hours of instruction, all students, both experimental and control received a post-test. The post-test had two parts. The first was a series of multiple choice questions similar to those in the pre-test. Essentially this part of the post-test was virtually identical to that of the pre-test in that the same concepts were tested, but with different questions. The second part of the post-test was a performance task in which students were presented with a scenario involving a toxic spill and they had to develop a plan for cleaning it up.

4.3 Results

There were two types of research questions of interest to the present study: 1) how good is INKS at modeling the knowledge relevant to solving problems, and 2) does an instructional framework based on the INKS framework lead to improved problem solving performance?

The answer to the first question is based on data collected by the computerized assessment tool which was actually developed under a separate project but was tested in the present study. To address this question, two types of correlations were computed.

The first was the correlation between concept knowledge as measured by the concept questions in the pre- and post-tests and the solutions to the application knowledge as measured by these two tests. To compute this, the number of correct concepts was tabulated for each student as well as the number of applied questions answered correctly. These were broken out by condition (experimental, control) and type of test (pre- or post-). There were too few observations to break the data out by knowledge type (e.g., procedural, causal), although this variable may be worth investigating in a future study. Table 1 below shows these correlations.

	pre-test	**post-test**
exper	.51 * (13)	.88 ** (13)
control	.54 ** (21)	.43 * (18)

Table 1: Correlations between conceptual and problem solving knowledge

numbers in parentheses are degrees of freedom

* $p < .05$

** $p < .01$

Note: some data missing due to computer network malfunction

It is interesting to note that the highest correlation is in the post-test of the experimental condition, which is the only cell in which students had been exposed to the subject matter. One explanation from this is that exposure to the subject matter reduces the pool of knowledge students will draw on to solve a problem. In other words, if students are unfamiliar with a subject area, the range of concepts they might draw upon to solve a problem is greater than that if they have been taught to solve problems in that area. Therefore, the correlation between a given set of concept knowledge and the ability to solve problems that use those concepts may be less when those concepts are less familiar than when they are more familiar. It turns out that the correlation of .88 is significantly higher than the other three ($p < .05$), which in turn are not significantly different from each other.

The other aspect of the research question regarding how good INKS is in predicting problem solving performance is looking at whether the change in concept knowledge as measured from pre-test to post-test correlates with the change in problem solving performance as measured from pre-test to post-test. To compute this, pre-test/post-test change scores for both of these measures were computed for the experimental group only (it was assumed that any change in scores in the control group would reflect variability in test-taking performance not true learning). The correlation obtained was .78 (df = 13, p < .01), suggesting that not only is INKS good at measuring current knowledge but also growth in knowledge.

The second research question this study addressed was whether an INKS-based instructional approach would lead to enhanced problem solving skills. To measure this, the performance task was used as it is currently the educational benchmark for subject area mastery. As is common in performance assessment, a four point scoring rubric was constructed to evaluate each student's response. Responses were evaluated by rater blind to experimental conditions and hypotheses. Table 2 presents mean scores on the performance task broken down by experimental. 9th grade control and 12th grade control conditions.

9th Grade Control	12th Grade Control	Experimental
1.55	3.00a	3.06a

Table 2: Mean Solution Scores on the Performance Task
Note: numbers not sharing a common coefficient are significantly different at the .05 level

Table 2 illustrates that the experimental group actually scored the highest of all groups on the performance task, although their performance was not significantly different from the 12th grade control group. However, the results show that an at-risk student population made up primarily of 9th graders (there were two 10th graders in the group) performed at a level significantly higher than their 9th grade mainstream counterpart and at a level comparable to mainstream 12th graders.

4.4 Discussion

The results of the present study lend support to the concept of INKS as a framework for modeling student knowledge and using it as a basis for developing instructional programs to build expertise. The above results suggest that an INKS model of student knowledge is highly predictive of problem solving performance, particularly in cases where students have been instructed in the subject matter. Additionally, the results suggest that changes in knowledge (i.e., learning) as measured by INKS is highly predictive of changes of performance.

Collectively, these results lend support to the general premise of the current research that problem solving is integrative in nature and that by using an knowledge representation framework that captures integrated knowledge, one can predict how well a person can solve problems.

The second major hypothesis of the present research was that an instructional framework modeled after the INKS representation scheme would be effective in building problem solving skills. The results of the present study confirmed this as well in that at-risk 9th graders performed at a mainstream 12th grade level at the end of instruction.

5.0 Intelligent Tutoring System Architecture

The second major goal of the present work was to develop an intelligent tutoring system architecture based on the INKS-based instructional methodology. Therefore, our methodology for building intelligent tutoring systems (ITSs) needs to be modeled after the six step methodology outlined at the beginning of this section. This ITS methodology, therefore, needs to specify not only an ITS architecture itself, but the tools required to support the other steps.

In order to devote as much of the project resources as possible to the development of an ITS architecture and supporting tools, the decision was made to adopt a testbed for which the knowledge elicitation (step 1 of the process) had already been conducted. Therefore the testbed chosen was algebra. In order flesh out the ITS methodology, the following technical components/ tools were developed:

1) an INKS knowledge representation structure
2) an INKS entry tool so that non-computer scientists could build INKS structures
3) a scripting language to present lessons
4) a knowledge assessment tool (built largely on another project) to assess student learning
5) an integrating architecture to link the assessment of student knowledge using INKS to appropriate instruction and presentation of that instruction.

Below we describe the overall architecture in general terms. We enclose three documents as an appendix that provides additional detail regarding the components. The first is the overall architecture that integrates the student model, the curriculum and associated activities, and the presentation of the lessons (including the user interface). The second document describes the scripting language that was developed (although not completely finished) called Gilligan. Gilligan controls the presentation of lessons to the user and the incorporation of student responses into the student model. The third document is a user's manual for Gilligan.

5.1 Generic ITS Architecture.

As discussed earlier, the second goal of the project is to develop a generic architecture for building ITSs. Figure 2 illustrates the general architectural design. We discuss this design in terms of its components.

The core activity of our architecture is assessing the student's knowledge and using this information to focus student activities for maximum learning. What make this architecture intelligent is that the choice of activities are directly based on what the student knows instead of being pre-programmed. The heart of the SIAM architecture is the interplay between the presentation manager, the student profile manager (student model) and the activity selection.

The activity selection module has two components: a course map that outlines the natural progression of topics to be taught

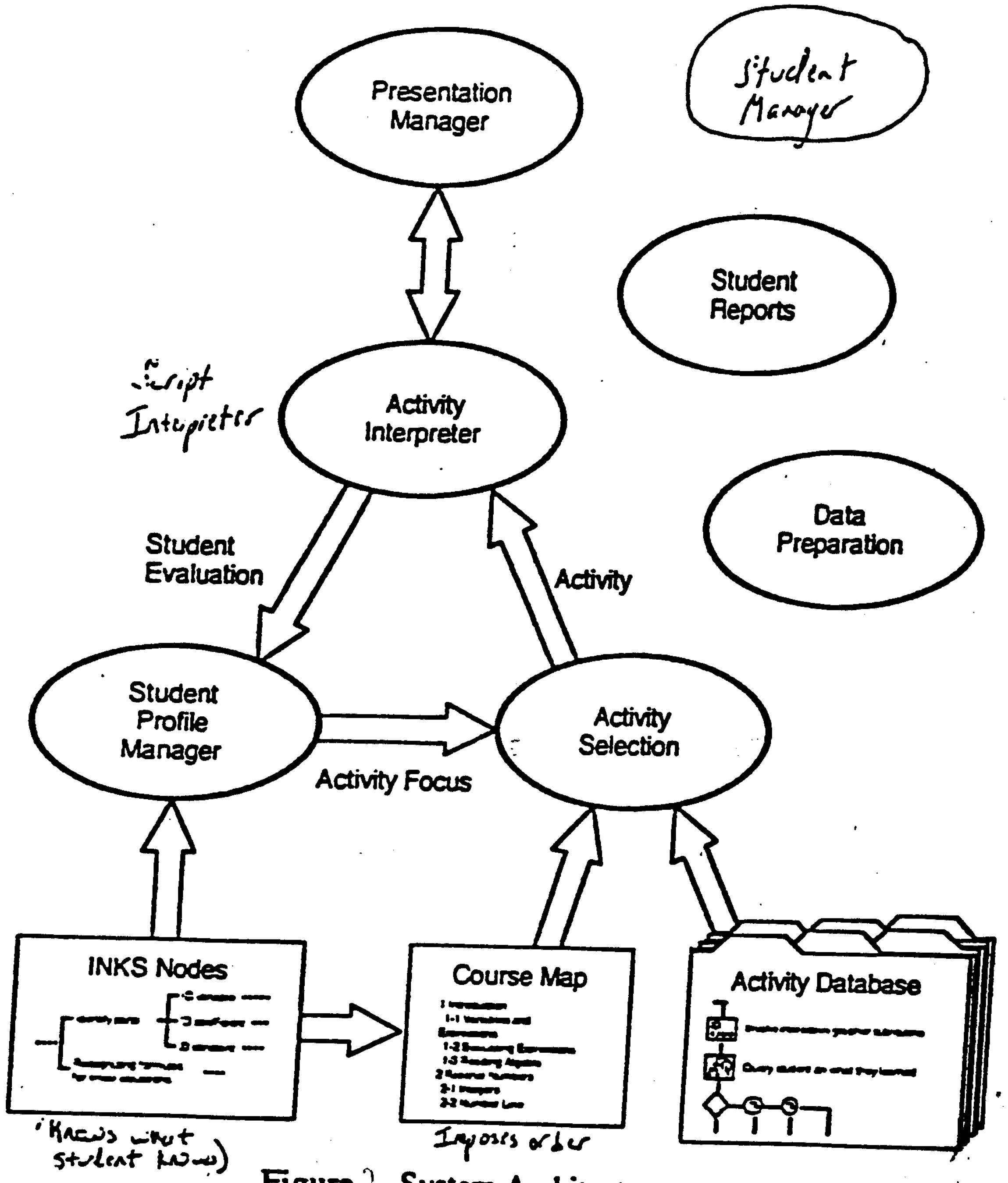


Figure 2. System Architecture

and a set of activities that are suitable for teaching particular topics both regular and remedial. The student profile manager influences which of these particular activities are selected based on the INKS model of the student. Activities are selected to address particular learning needs. In the case where more than one activity is appropriate, the activity is selected that satisfies the most learning needs. Once the activity is selected, it is presented to the student. The Gilligan scripting language serves as the interface between system and student. It both directs what the student sees on the screen and maps the student responses to the appropriate portion of the INKS so that the Student Profile Manager can update the student model and determine what the new learning needs are.

The SIAM architecture includes supporting tools. An INKS entry tool allows a designer to enter INKS content into a Fox Pro database so that it can be used by the system. The interface to this tool allows the user to type in the content for each slot in the INKS node so the user does not have to worry about the structure of the INKS itself, only the content of the knowledge. An INKS viewing tool allows the user to see the INKS structure in graphic format so that s/he can check the INKS content for accuracy and validate that concepts are correctly integrated with each other.

As discussed earlier, CSA is a tool for providing additional validation regarding what someone knows about a domain by presenting direct probes as opposed to observing only their problem solving behavior. CSA is integrated via an assessment tool that is connected to the student model. When an error state is triggered within an INKS node, a set of probes is generated to validate the error. This ultimately can influence what remediate a student may get.

The rest of this section describes the intelligent tutoring system (ITS) architecture and explains some of its functions.

5.1.1 Activities

One core concept in the ITS architecture is that of an *activity*. Activities control all interactions with the student. Each activity is made of *tasks*, which are able to present text and graphics, animate graphics, engage in interactive exploration of the information, query for information, assess student responses, and chose future activities. Simple activities may have one task such as the display of text to explain a concept. Complex activities may have full simulation-scenarios or multimedia presentations with multiple decision paths based on student responses.

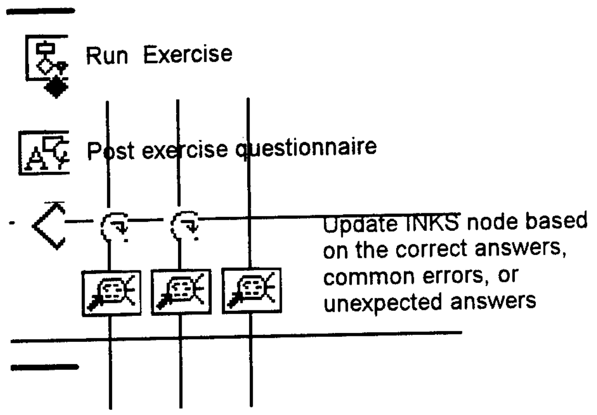

Run Exercise
Post exercise questionnaire
Update INKS node based on the correct answers, common errors, or unexpected answers

Figure 3 An example activity line

The tasks are connected together on an *activity line*. Figure 3 shows one such line. Activities flow from the top to bottom and the horizontal bars at each end represent the start and stop of the activity. Different icons are used to represent different classes of tasks. In the example, the first icon tell ITS to call another activity line (a subroutine) and when that line is finished return to this line. The next icon writes a question on an interaction window and prompts the student for a reply. It is followed by hollow diamond which intercepts the student's reply and interprets their choice. Each potential choice is shown as a arrow surrounded by circle and for each reply some new path is taken. If the choice is not one of the expected choices, the final (right most) path is taken. Below each choice is a icon that places evidence on the INKS node (explained later). This evidence will help determine what the student knows and where remediation should take place.

Instructional technologists break their activities into pretests, lessons, practice (problems with immediate feedback), tests and remediation (explaining the subject in a different manner). As can be seen, activities forms a powerful programming language that allows the instructional technologists to perform all of these activities and to build complex lessonware.

The description to this point has the instructional technologist explicitly controlling what activities are being shown at each step of the instruction. This lacks flexibility. Ideally, we want to assess student understanding and, at appropriate places in activity flow, choose remediation activities. This is at the heart of our intelligent tutoring. The bottom three icons in Figure 3 are being used to collect data on the student's understanding and use this evidence to modify INKS nodes. For this evidence to be exploited, the instructional technologist must identify points in instruction where remediation would be appropriate.

In Figure 4, a remediation opportunity can be seen on the first icon. The solid diamond at the bottom of the icon informs ITS to remediate at this point if the student did not understand some aspect of graphing (what the task taught).

To remediate, the ITS temporarily puts the current activity line aside and begins the chosen remediation activity. Remediation activities will continue to be chosen until the student shows understanding in the material. Once they understand, the instructional flow continues where it left off. Remediation is recursive and will delve as deep as necessary to teach the material.

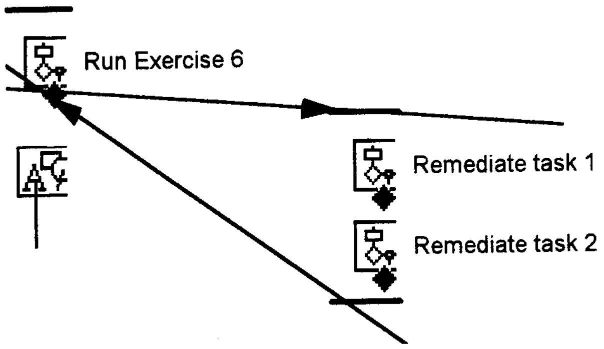


Figure 4. A remediation activity

All activities connected directly to INKS nodes by a *statement of objectives.* These identify the competencies or misunderstandings are addressed by this activity. The required list of competencies or misunderstandings is derived from an analysis of the INKS description. Activities can address multiple competencies and misunderstandings. The system will always attempt to select activities that best match the students needs without repeating activities unnecessarily. Activities should also be chosen that adapt to different student learning styles and problem-solving methodologies. The system will also attempt to exploit different methods of presentation.

5.1.2 INKS

To master a subject, the student must learn and apply many different types of knowledge. The *Integrated Knowledge System (INKS)* is an organizational methodology that groups and synthesizes these knowledge sources into one common description. This is another core concept used by ITS. It allows the different facets of problem knowledge to be used holistically and gives an educational system considerable flexibility in dealing with the student and the errors they make during learning.

INKS knowledge is represented by a network of information nodes. Each node describes the knowledge necessary to perform one educational task. Figure 5 presents a pictorially representation of an INKS node. The *node name* identifies this node and is unique from all others.

The *mental model* describes why this node is used.

The *input scene* specifies when this node can be applied to the task. It is a set of preconditions that are expected to be true before the knowledge in this node can be applied.

The *procedure* specifies the set of steps that must be achieved to perform this task.

The *output scene* is the expected result from applying the procedure.

Background concepts are those things implicitly assumed to be known by the student. These concepts are declarative in nature and differ from input scenes in their focus. The input scene and mental model describe when and why knowledge should be applied. The background concepts are those things of a more general nature that may be used to reason in the problem domain.

The *state table* holds information on the student's performance and errors they are making. Each INKS node has one *correct state* (the student know this knowledge), zero or more common error states, and one *unknown error state* (when the student makes an error that is not explicitly represented as a state). The different states are updated as the student is asked questions and responses.

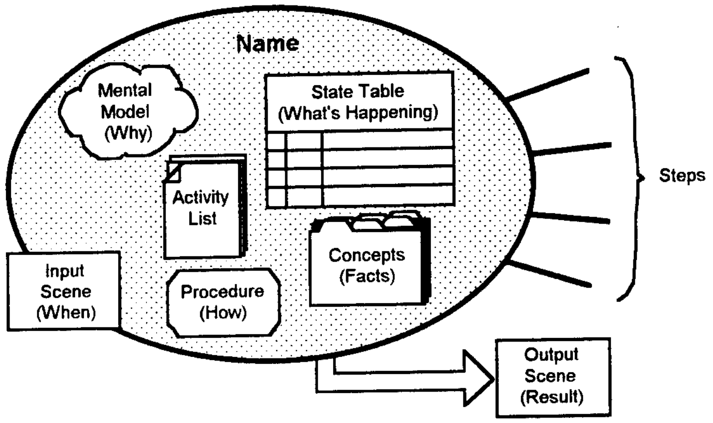


Figure 5. An INKS node

The INKS *activity list* lists all activities that can teach one or more aspects of this node. Each node must have an activity for the mastery standard state and one for each error state. All important concepts have multiple activities which either explain the concept from a different perspective or are designed to address common errors made by students.

6.0 Discussion and future directions.

The present effort demonstrated the feasibility of the INKS framework as the basis of a methodology for developing ITSs, both from an instructional and a technological perspective. Clearly, further research is needed to complete the ITS architecture and supporting tools so that trainers who do not specialize in ITS development can construct ITSs for practical use.

In addition to the completion of the development and testing efforts begun in the present work, we would like to discuss three extensions of ITS technology that we believe would make a significant contribution to the field. We believe our INKS-based ITS framework could support developments in each of these areas.

We have discussed earlier that the typical rule-based ITS is very constraining on the behavioral paths it allows a user to take. Any action that deviates from the expected path is flagged as an error and the user is instructed to substitute his/her "incorrect" response with a "correct" one. While this methodology can be useful in teaching formalized procedures, we argue that it has little ecological validity for the wide range of ill-structured, open-ended problems that people face in life.

We have often made the point, for example, that this model of ITSs could not be used to teach people how to build ITSs because there is no single "correct" procedure for how to build one. Most problems have a variety of solutions that are roughly comparable in their acceptability and often, they are too many to enumerate and program into an ITS' expert knowledge base.

Further, it is often difficult to tell from a single step in a procedure whether a person is making a mistake or on the right track. Often, it is a sequence of steps that are necessary to infer whether a person's solution path is appropriate. A good example of this is chess. There are 20 possible first moves for the white pieces in any game. Well over half of these moves have formalized opening systems structured around them, typically with extensive branching to allow a myriad of "acceptable" move sequences. Therefore, it is impossible to judgment the goodness of most of these moves unless they are viewed in the context of the overall strategy of the subsequent play. Such an occurrence is problematic for a rule-based ITS that seeks to evaluate each action for its correctness at the time the action is made.

Related to this issue is the phenomenon of exploratory learning. Here, the student is expect to learn by exploring a problem solving environment, collecting information, taking actions, and discovering the consequences of those actions. Exploratory learning is currently a popular paradigm within the educational and training communities. Therefore, it is valuable for ITS technologies to support this type of learning.

One valuable method for utilizing an exploratory learning environment an a computerized setting is through games. Games naturally lend themselves to exploration. Further, the fact that they are fun induces people to spend substantial amounts of time (most willingly too!) playing them. By embedding intelligent tutoring within a game context, one can capitalize on user motivation to spend time and master the environment.

Like exploratory learning environments, games typically support a great deal of user control and freedom of action. As was discussed above, this tends to be problematic for rule-based ITSs that expect users to follow fixed action sequences.

In order to build exploratory learning and open-ended action into an ITS, the primary technical challenge appears to be giving the ITS the ability to understand user behavior when it does not follow a preprogrammed action pattern. This has tremendous ecological validity given the tremendous variability in the way people naturally solve problems in the real world. This is also reflected in real world teaching as students typically are not tightly constrained in how they solve problems. Therefore, teachers need to possess the ability to view a workproduct and make inferences as to the problem solving process used by the student.

The INKS-based ITS framework offers an approach to address this issue. Higher level structures in the INKS framework specify top-level goals and the types of actions necessary to accomplish these goals. These actions are organized based on their causal enablements to the goal rather than a specific order in which they must be carried out. The problem solving structures would then contain the goal, the solution constraints, necessary subgoals/steps and any steps that are dependent upon each other (which would dictate sequencing requirements).

With such a problem solving model, it becomes less relevant what order steps are conducted as long as logical enablements are met. For example, suppose one wanted to teach scientific problem solving in the context of a detective game (detective work is much like scientific research as both involve hypothesis testing through the collection and analysis of empirical data). The tutoring component might have the following constraints: it is necessary to collect sufficient evidence to support the favored hypothesis (e.g., convict the suspect) and rule out competing hypotheses (e.g., show that other suspects could not have done it). With these general constraints, the specific order in which a student might collect data becomes less relevant as long as the student fulfills both requirements. Therefore, such an INKS-based approach could support open-ended, exploratory learning.

A second future direction the present research could take represents a marriage of intelligent tutoring systems with question answering systems that use natural language. The basic motivation behind such a system is based on the everyday observation that sometimes people learn by simply asking questions and having something explained to them. Currently, ITS research has emphasized presenting lessons and teaching procedures rather than instruction through "dialog".

It is important to emphasize that traditional question answering systems are not, in and of themselves, ITSs nor can they serve in that capacity without significant enhancements. A good teacher does more than simply convey information. Teachers recognize that there are educational goals that may extend beyond simply the scope of the information being requested by the questioner. Therefore, a teacher may provide more information than has been requested or even answer a different question if she determines that the student's question is directed toward the wrong educational objective.

A good teacher also assesses the questioner's current level of knowledge and ability to assimilate new information. For example, if a novice asks "tell me about the work you do in intelligent tutoring systems", a wise response will cover basic level topics, whereas if a fellow researcher asks the same question, one would expect a much more sophisticated and technical response. Related to this, a good teacher will gauge whether the student appears to understand the answer to a query and may often attempt to explain the concept in several ways and over time, will develop a model of what types of explanations (e.g., concrete example, analogy) work best with what types of students.

A good teacher will also ask questions themselves as part of the teaching process in order to assess what background a student has, whether they are understanding the answers being given them or even to clarify what question the student is really asking in case there is more than one interpretation or more than one possible answer, the relevance of which depends on the student's goal. A good teacher will anticipate the student's overall educational goal and may offer information the student has not requested if she deems that the student might find it interesting, e.g., telling the student of related topics.

Finally, a teacher will keep track of topics already discussed and draw upon them and build upon them in further discussions.

These are extremely interesting topics for further research and ones with tremendous practical application. For example, many software packages have help files or users manuals. Rather than having a user struggle through these, it is much more efficacious for a user to simply ask the system a question. Further, "virtual reference librarians" for such things as the Internet (or even mundane things like libraries or museums) could help people navigate through vast resources that are available but difficult to anticipate by the user (so "intelligent" questions may be hard to generate).

Preliminary research related to these issues has been done by Johanna Moore (1995). She has developed discourse-based intelligent tutors that interact with students through dialog. Students are able to ask the tutor questions and receive answers. The tutor has the capability to refer to previously discussed material or to allow the student to request an explanation or elaboration of an answer.

Missing from Moore's work is student modeling. In other words, the Moore's tutors do not evaluate what a student knows compared to an educational goal and then direct the dialog to fill in missing knowledge. Moore's tutor (which is an enhancement of Sherlock) relies on the student to direct the queries.

We believe our ITS can make a valuable contribution to this type of technology. The INKS framework can be used as a model of both students and expert knowledge, which can help drive the discourse. Additionally, our CSA elicitation technique could form the basis of probing questions that seek to evaluate what the level of student knowledge is and what information is most needed.

One of the principal needs that have been expressed by members of both the educational and training communities is for technology to support group training. Currently, we have intelligent tutoring systems that support training of individuals according to their specific needs and simulation environments that support group problem solving, but without individual feedback and instruction. A major technical challenge is to bridge these two technologies so that one can enjoy the benefits of group problem solving training (which is more ecologically valid than individual problem solving for most real world tasks) and still maintain the advantage of personalizing instruction to optimize the learning of each participant.

We believe that one of the main challenges in team training is to accommodate the diverse reasoning styles associated not only with specific students being trained, but also with specific tasks. For example, in our research we have found that intelligence analysts need to be data-driven, whereas operations officers need to understand high level plans (Leddo et al., 1990). Therefore, an intelligent tutoring framework that purports to be a team trainer must be able to accommodate these different knowledge styles. In order to do this, its knowledge framework must be sufficiently rich to handle the different types of knowledge associated with different individuals and tasks.

But there is a more subtle requirement that is important here. It is not enough for an intelligent tutoring framework to support the training of individual skills. This would imply a series of independent trainers. Rather, if the goal is to promote enhanced team effectiveness, the requirement is to train individuals so that their performance enhances the team. This means that any intelligent tutoring framework must be able to integrate the knowledge associated with specific tasks, evaluate it against an overall team standard, and then teach at the individual level in accordance with this team standard. In other words, the framework needs to be able to reason about the individual subject areas integratively using both inductive and deductive knowledge processing.

We believe the INKS framework that we have developed and used on the present project provides an excellent framework to deliver team training. The INKS framework itself was developed in large part based on research on decision makers in different tasks and how they work together. We now discuss how the INKS framework can be used in team training.

In earlier sections of this report, we discuss our intelligent tutoring framework. The first step is to build expert knowledge models of the domain being taught, typically through knowledge elicitation sessions with domain experts. These knowledge models can be supplemented by observation of the experts on the job and through training materials. For a team training intelligent tutoring framework, it would be essential to think about building the knowledge model on two levels.

The first is in terms of high-level tasks. This knowledge model needs to emphasize the overall team goals, the (mental) model of who the team members are and how they interact, and the strategic plans that the team needs to carry out. This might also be the type of knowledge that the team leader would have, who in the absence of a tutor would be responsible for the performance and corrective instruction of the team.

The second level of the knowledge model would deal with the specialized knowledge that individual team members have.

Typically, (and this is especially true in military teams) teams are comprised of a team leader, who is a generalist in terms of his/her knowledge and a core of functional specialists. This is true whether we are talking about a combat team that may have a commander, intelligence officer, logistics officer, personnel officer, etc. or a medical team that may have a surgeon, anesthesiologist, nurse, etc.

We would expect these two levels of knowledge to be hierarchical. At the lower levels of the hierarchy, we would expect the functional knowledge to consist largely of semantic (content or factual) knowledge and procedural (production rule) knowledge that supports performing the specific functions. Above this knowledge at a higher level of abstraction, we would expect the integrative knowledge to consist largely of script-based knowledge that embodies the goal and planning knowledge of the team and mental models to integrate the functional lower level knowledge with respect to how they relate to the higher level plans.

We do not believe that any of the single knowledge structure frameworks that currently dominate intelligent tutoring system research would suffice in this role. The most predominant knowledge representation framework in ITSs is production rules. As discussed earlier, production rules are temporally ordered procedures. Production rules lack the hierarchical structure necessary to integrate lower level bodies of knowledge. They work effectively in teaching such things as computer programming (in which the output is itself a set of procedures) but fail in more planning-oriented problem solving instruction such as geometrical proofs.

The second most common single knowledge structure framework is frames (cf., Johnson and Soloway, 1985). While frames are more schematic in nature than production rules and therefore are better at integrating lower level knowledge, frame-based knowledge is typically generic in nature and does not deal well with rich, diverse situations such as the types that comprise medical emergencies or combat scenarios.

We now briefly discuss how our INKS-based ITS framework would handle team training. The first step that would be necessary would be to define the team goals and component tasks. This would include identifying the relevant team members to be trained and the training priorities. We would perform this in conjunction with the end users and relevant subject area experts.

The next step would be to form an INKS model of effective team problem solving. This provides a performance standard that the training would be designed to support. As an aside, this is a common practice in educational settings as a prior step to developing curricula. Once the performance standard has been identified, the next step is to build a knowledge model of the team skills that are necessary to reach this standard. At this point the knowledge model is high level and includes such things as the subgoals or substeps that are necessary to be performed in order to achieve the team goal, identifying the personnel responsible for achieving these subgoals and how these subgoals combine to achieve the overall objective.

This model then will drive the instructional objectives for each individual member of the team. The instructional objectives focus on the necessary team member output and the knowledge necessary to achieve this output. At this point, INKS knowledge models of the different functional specialists are developed so that specific team member instruction can be generated. Once these INKS structures are developed, lessons, including those presented by simulations are developed.

The problem solving/simulation-based training would focus on two things. First, the problems need to strengthen the individual skills of the functional specialist. Here, the problems would focus on mastering the basic concepts and procedures needed by the functional specialist. In essence, the math content of the present ITS can be considered an example of this. However, once these skills are developed, it is crucial that they are placed in the context of team problem solving and performance.

Therefore, the second focus of simulation-based instruction is on team problem solving and performance. Here, the student practices his or her skills in the context of a team. In this context, either the ITS itself can play the role of the other team members or several students can practice together, each playing the role of a different team member. The advantages of the ITS playing the other team members is that this gives the instructor greater control over the types of scenarios and actions that the student will witness and have to respond to. It also creates the opportunity for students to see the "correct" procedures being carried out by the team members. The advantages of having multiple students practice is that the situations they encounter may be more realistic in that human variability and error are more likely to be present in the human participant variation than if the ITS plays all roles. Another advantage, particularly if the students being instructed will also work with each other in the "real world" is that they can develop a group dynamic and work out any problem in a simulated world rather than working out their mistakes as a team in the real situation. One drawback of having a team comprised of student, particularly if the students are rough, is that students may develop bad habits working with people who are error prone.

We believe that students should probably gain experience in more controlled simulations, perhaps those directed by the ITS or having an instructor be a participant. Once the student is competent at his or her functional skills, other students can be introduced to play other members of the team.

There are two more issues that we would like to raise with regard to team training. First, when a person goes from being an individual problem solver to a member of a team, there are other necessary skills beyond simply the technical skills of one's job description. For example, questions of resource allocation come into play (among other things related to the logistics of the group) as well as group dynamic skills such as the ability to work with group members who have different cognitive and problem solving styles and the ability to communicate with group members. In fact, so important is the skill of communication that in one study, experienced analysts rated the ability to communicate one's conclusions as the single most important factor in determining how effective they were (Martin, Mullin and Leddo, 1989). Therefore, we believe that an important part of training group problem solving includes emphasizing these skills as well.

The second issue worth raising is assembling the team for training. In many instances, a group can be assembled in a single location at a single time for training. However, today's military is situated in diverse locations and an important need is to replace the traditional schoolhouse model of collocating trainees with a methodology that allows trainees to receive instruction without taking time from their units. Another good example of this are reserve personnel. Here, the people who require training are only "assembled" for short periods of time each year. There is an important requirement for personnel to sustain their training throughout the course of the year, particularly since it may be unclear when such people will be called on to use such training.

One solution for delivering such training may be through the use of distance learning technology involving teleconferencing, or perhaps more economically, training via the Internet. Below we present some initial thoughts as to how such training may be accomplished.

From its inception, the academic community has been very active on the Internet and many of the services currently available are educational in nature. Access to these services have excited educators from all walks of life (e.g., public schools, universities, and corporate trainers). The availability of Mosaic (discussed below) have only added fuel to the fire. Many educational systems are either connected to the Internet or have pilot programs to assess its feasibility. This discussion will begin with a brief discussion on the Internet and Mosaic and finish describing how it supports future education opportunities.

Built into the fundamental design of the Internet is an ability for new services to be created and provided to end-users. Recent years have seen an explosion of these services. While they have profoundly expanded the usefulness of the Internet, users are increasingly aware that services exist that could assist them, but they do not know how to locate these services or how to use the service once located. A new class of program has emerged which assist users navigate the Internet and locate relevant services. The Mosaic system from the National Center for Supercomputer Applications (NCSA) at the University of Illinois is one example.

Mosaic begins by presenting a single hypermedia interface to the user. Once learned, the user can access all supported services from this interface. Mosaic handles the actual interactions with the underlining services and once a reply have been acquired, Mosaic uses a data appropriate viewer to present the information. By directly handling the underlining service, Mosaic insulates the user from the idiosyncratic requirements and behaviors of different services. Mosaic's overall effect is that of a very large hyperlinked document that covers any and all topics.

Since our software architecture separates lesson presentation from lesson management, we are able to create a new Internet service that uses Mosaic for lesson presentation. This architecture has four advantages.

First, by using Mosaic for lesson presentation, our software would have access to any presentation service available to Mosaic. This currently includes text, graphics, sound and video. Using Mosaic allows us to build multimedia lessons without having to build multimedia presentation engine.

Second, users of Mosaic can use our software without having to learn a new user interface. Learning new software interfaces is an investment of time and energy that many people avoid unless they perceive the benefit much greater than the cost. In Mosaic's case, many Internet users have already made this decision and understand how to use Mosaic. By using Mosaic, one can capitalize on the training these users have already done.

Third, this approach makes our educational software available anywhere Internet services exist. Indeed, students could access the software from anywhere in the world. This is one version of distance learning.

Finally, by using the Internet (implicit in use Mosaic), collaborative learning can be supported. Since many skills require a team to perform the task, training for these skills often require the complete team for training. This is often difficult when members live in different geographic areas. Since the Internet is available anywhere in the world, team training can occur even when members are geographic separate.

Our approach is also useful for integrating digitized video. Video materials are often used to augment other instructional techniques. Much of this material is of high quality and should be reused. It is easily integrated into our architecture. For each video clip, a lesson is created listing the video's objectives and describing a method for student evaluation. Once these lessons are placed in the curriculum, they will be selected in the same manner as any other lesson. Indeed, if the student shows preference for video material, these lessons will get preference.

Other, non-computer lesson material, can be supported in a manner similar to video. For each lesson using non-computer materials, a lesson is created listing the lesson objectives and describing a method for student evaluation. The software architecture will automatically select these lessons when appropriate. The types of non-computer lesson material can be very broad and two example curriculum will be listed below.

A curriculum attempting to teach mathematics or some other domain may have a student (or whole classroom) solve a problem and record the results. The computer-related curriculum would then assist students by posing questions on their results and assisting them in answering these questions. Indeed, if the students are connected by the Internet, they could exchange results with students all over the world and do very interesting and complete analysis.

The above discussions are meant to illustrate the tremendously rich potential of simulation-based ITS work. We have only begun to scratch the surface of marrying instruction with technology. We believe future work in this area holds great promise.

REFERENCES

Anderson, J.R. (1980). Cognitive psychology and its implications. San Francisco, CA: W.H. Freeman and Co.

Anderson, J.R. (1982). Acquisition of cognitive skill. Psychological Review, 89, 369-405.

Anderson, J.R., Farrell, R. and Sauers, R. (1984). Learning to program in LISP. Cognitive Science, 8, 87-129.

Anderson, J.R., Boyle, C. and Reiser, B. (1985). Intelligent Tutoring Systems. Science, 228, 456-462.

Bloom, B.S. (1968). Learning for mastery. Evaluation Comment, 1(2).

Bloom, B.S. (1984). The 2-sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. Educational Researcher, 13(6), 4-16.

Bower, G.H., Black, J.B., and Turner, T.J., Scripts in memory for text. Cognitive Psychology, 1979, 11, 177-220.

Brna, P., Ohlsson, S., and Pain, H. (1993) (Eds.) Artificial Intelligence in Education, 1993. Association for the Advancement of Computing in Education.

Chi, M., Feltovich, P. and Glaser, R. (1981). Categorization and representation of physics problems by experts and novices. Cognitive Science, 5, 121-152.

de Kleer, J. and Brown, J.S. (1981). Mental models of physical mechanisms and their acquisition. In J.R. Anderson (Ed.), Cognitive skills and their acquisition. Hillsdale, NJ: Erlbaum.

Ericsson and Simon (1984). Protocol analysis: Verbal reports as data. Cambridge, MA: The MIT Press.

Galambos, J.A. (1986). Knowledge structures for common activities. In J.A. Galambos, R.P. Abelson, and J.B. Black (Eds.) Knowledge structures. Hillsdale, NJ: Erlbaum.

Galambos, J.A., Abelson, R.P. & Black, J.B. (Eds.) (1986) Knowledge Structures. Hillsdale, NJ: Erlbaum.

Greer, J. (1995) (Ed.), Artificial Intelligence in Education, 1995. Association for the Advancement of Computing in Education.

Johnson, W.J. and Soloway, E. (1985). PROUST: An automatic debugger for Pascal programs. BYTE, 10, 179-190.

Johnson-Laird, P.N. (1983). Mental models. Cambridge, MA: Harvard University Press.

Laskey, K.B., Leddo, J.M., and Bresnick, T.A. Executive thinking and decision skills: A characterization and implications for training (Technical Report 89-12). Reston, VA: Decision Science Consortium, Inc., September, 1989.

Leddo, J.M., Cardie, C.T., and Abelson, R.P. An integrated framework for knowledge representation and acquisition (Technical Report 87-14). Falls Church, VA: Decision Science Consortium, Inc., September 1987.

Leddo, J.M., Mullin, T.M., Cohen, M.S., Bresnick, T.A., Marvin, F.F., and O'Connor, M.F. Knowledge Elicitation: Phase I Final Report, Vols. I & II (Technical Report 87-15). Reston, VA: Decision Science Consortium, Inc., May 1988.

Leddo, J.M., Sak, S.G., and Laskey, K.B. THINKER: An intelligent tutoring system for building expert knowledge (Technical Report 89-7). Reston, VA: Decision Science Consortium, Inc., April 1989.

Leddo, J. and Cohen, M.S. Cognitive structure analysis: A technique for eliciting the content and structure of expert knowledge. Proceedings of 1989 AI Systems in Government Conference. McLean, VA: The MITRE Corporation, 1989.

Leddo, J., Cohen, M.S., O'Connor, M.F., Bresnick, T.A., and Marvin, F.F. Integrated knowledge elicitation and representation framework (Technical Report 90-3). Reston, VA: Decision Science Consortium, Inc., January, 1990.

Leddo, J., Laskey, K., and Vane, R. Intelligent tutoring games for interest-based learning. (Technical Report 92-2). Reston, VA: Research Development Corporation, October, 1992.

Leddo, J., Campbell, V., Black, K., and Isaacs, C. Knowledge elicitation tools for educational evaluation. (Technical Report 92-1). Reston, VA: Research Development Corporation, March, 1992.

Leddo, J. and Sak, S. Knowledge Assessment: Diagnosing what students really know. Presented at Society for Technology and Teacher Education, March, 1994.

Lesgold, A., Lajoie, S.P., Bunzo, M. and Eggan, G. (1990). A coached practice environment for an electronics troubleshooting job. In J. Larkin, R. Chabay and C. Cheftic (Eds.), <u>Computer assisted instruction and intelligent tutoring systems: Establishing communication and collaboration.</u> Hillsdale, NJ: Erlbaum.

Martin, A.W., Mullin, T.M., and Leddo, J.M. Learning from experienced analysts (LFEA) phase zero: Results of working conference on expertise in intelligence analysis. Reston, VA: Decision Science Consortium, Inc., May 1989.

Minsky, M. (1975). A framework for representing knowledge. In P. Winston (Ed.), The Psychology of computer vision. NY: McGraw-Hill.

Moore, J.D. (1995). Discourse generation for instructional applications: Making computer tutors more like humans. In J. Greer (Ed.), Artificial Intelligence in Education, 1995. Association for the Advancement of Computing in Education.

National Assessment of Educational Progress. (1989). Crossroads in American Education. Princeton, NJ: Educational Testing Service.

Newell, A. and Simon, H.A. (1972). Human problem solving. Englewood Cliffs, NJ: Prentice Hall.

Polson, M.C. and Richardson, J.J. (1988). (Eds.), Intelligent Tutoring Systems: Lessons learned. Hillsdale, NJ: Erlbaum.

Quillian, M.R. (1966). Semantic memory. Cambridge, MA: Bolt, Beranek and Newman.

Schank, R.C. & Abelson, R.P. (1977). Scripts, plans, goals and understanding. Hillsdale, NJ: Erlbaum.

Schank, R.C. (1982). Dynamic memory: A theory of reminding and learning in computers and people. MA: Cambridge University Press.

Schoenfeld, A.H. and Herrmann, D.J. (1982). Problem perception and knowledge structure in expert and novice mathematical problem solvers. Journal of Experimental Psychology: Learning, Memory and Cognition, 8(5), 484-49.

Shavelson, R.J., Webb, N.M., Stasz, C. and McArthur, D. Teaching mathematical problem solving: Insights from teachers and tutors. In R.I. Charles and E.A. Silver (Eds.) The teaching and assessing of mathematical problem solving. Hillsdale, NJ: Erlbaum, 1988.

Shute, V.J. and Regian, J.W. (1990). Rose garden promises of intelligent tutoring systems: Blossom or thorn? Paper presented at the Space Operations Automation and Robotics conference, Albuquerque, NM.

Weiser, M. and Shertz, J. (1983). Programming problem representation in novice and expert programmers. International Journal of Man-Machine Studies, 19.

Presented Papers.

The following are papers that are based on the present research:

Leddo, J. and Gang, K. Integrating standard setting, assessment, instruction, technology, and staff development. Presented at the Virginia Middle School Association Conference. March, 1995.

Leddo, J. Assessing what students really know. Curriculum Product News. October, 1994.

Leddo, J. Did they learn anything?: Finding out with a knowledge assessment tool. Presented at the National School Board Association Techology and Learning Conference. October, 1994.

Leddo, J. Did they learn anything?: Finding out using knowledge elicitation techniques. Presented at the California Science Teachers Association Conference. October, 1994.

Leddo, J. and Sak, S. Knowledge Assessment: Diagnosing what students really know. Presented at Society for Technology and Teacher Education, March, 1994.

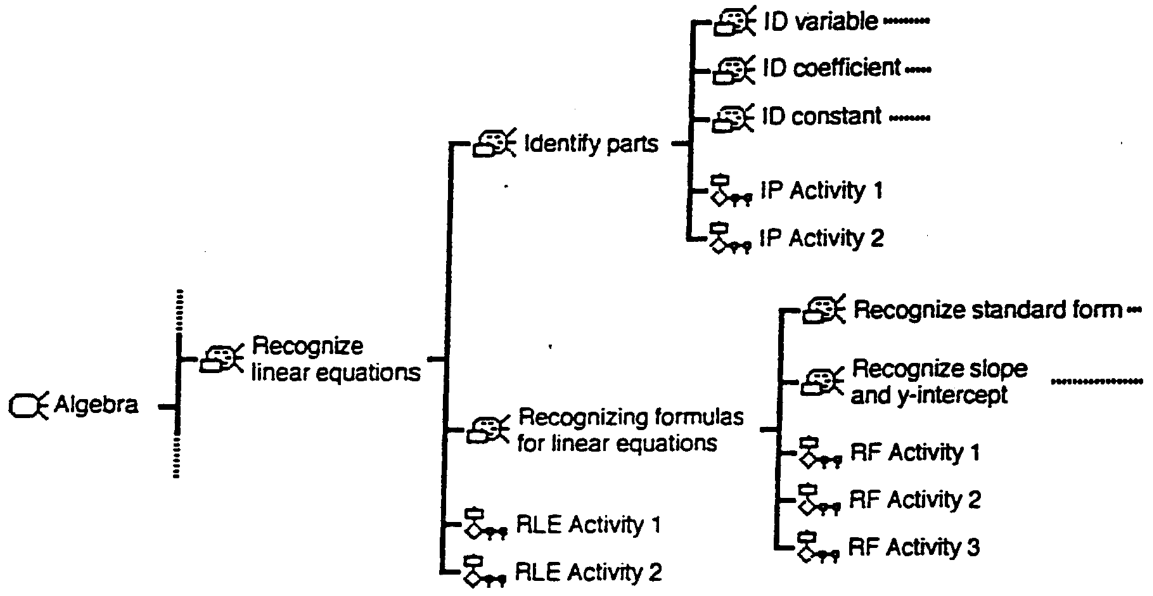
APPENDICES

SIAM Architectural Design Document

The Gilligan Authoring Language

Graphical Interaction Language for Lesson Integration, Generation And Navigation (GILLIGAN)--Users Manual

Siam
Architectural Design Document



This document has been brought to you by
the friendly members of your technical staff

Research Development Corporation Proprietary

TABLE OF CONTENTS

FIGURES

1 Introduction

This document describes the Siam system. Siam is RDC's core assessment and intelligent tutoring technology which is embedded in all of our products. The primary goal of this document is to provide needed design information. Section 1 (this section) and 2 (the overall architecture) should be understandable to others. Section 3 is a detailed description of the various modules that make up the system and is specifically for the RDC programmers. Appendix A is a glossary of terms.

The Siam system is an intelligent tutoring system. One unique aspect of Siam is that it bases student activities on an objective assessment of their individual understanding. In other words, this system adapts to each student and focuses its efforts to maximize learning.

Siam is designed to present multimedia materials. This include text, graphics, video, sound, and interactive games.

2 System Architecture

The core activity of our architecture is assessing the student's knowledge and using this information to focus student activities for maximum learning. What make this architecture intelligent is that the choice of activities are directly based on what the student knows instead of being pre-programmed. This differs from most educational systems whose curriculum must be executed in a specific order.

The rest of this section describes the Siam architecture and explains how it functions.

2.1 Activities

One core concept in the Siam architecture is that of an *activity*. Activities control all interactions with the student. Each activity is made of *tasks*, which are able to present text and graphics, animate graphics, present video or sound, engage in interactive exploration of the information (e.g. an interactive grapher or a game), query for information, assess student responses, and chose future activities. Simple activities may have one task such as the display of text to explain a concept. Complex tasks may have full multimedia presentations with multiple decision paths based on student responses.

The tasks are connected together on an *activity line*. Figure 1 shows one such line. Activities flow from the top to bottom and the horizontal bars at each end represent the start and stop of the activity. Different icons are used to represent different classes of tasks. In the example, the first icon tell Siam to call another activity line (a subroutine) and when that line is finished return to this line. The next icon writes a question on the interaction window and prompts the student for a reply. It is followed by hollow diamond which intercepts the student's reply and interprets their choice. Each potential choice is shown as a arrow surrounded by circle and for each reply some new path is taken. If the choice is not one of the expected choices, the final (right most) path is taken. Below each choice is a icon that places evidence on the INKS node (explained later). This evidence will help determine what the student knows and where remediation should take place.

Instructional technologists break their activities into pretests, lessons, practice (problems with immediate feedback), tests and remediation (explaining the subject in a different manner). As can be seen, activities forms a powerful programming language that allows the instructional technologists to perform all of these activities and to build complex lessonware.

The description to this point has the instructional technologist explicitly controlling what activities are being shown at each step of the instruction. This lacks flexibility. Ideally, we want to assess student understanding and, at appropriate places in activity flow, choose

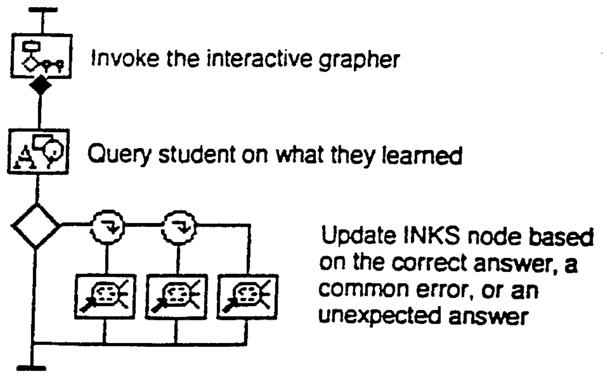

Invoke the interactive grapher
Query student on what they learned
Update INKS node based on the correct answer, a common error, or an unexpected answer

Figure 1. An example activity line

remediation activities. This is at the heart of our intelligent tutoring. The bottom three icons in Figure 1 are being used to collect data on the student's understanding and use this evidence to modify INKS nodes. For this evidence to be exploited, the instructional technologist must identify points in instruction where remediation would be appropriate.

In Figure 1, a remediation opportunity can be seen on the first icon. The solid diamond at the bottom of the icon informs Siam to remediate at this point if the student did not understand some aspect of graphing (what the task taught). Figure 2 shows this remediation activity taking place.

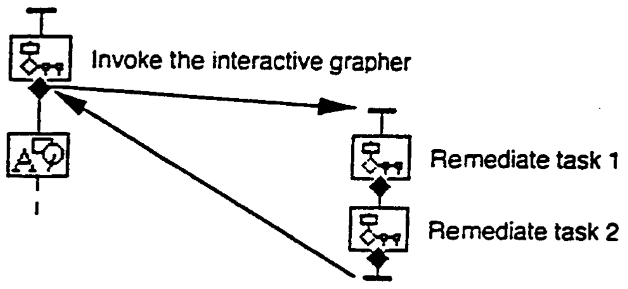


Figure 2. A remediation activity

To remediate, Siam temporarily puts the current activity line aside and begins the chosen remediation activity. Remediation activities will continue to be chosen until the student shows understanding in the material. Once they understand, the instructional flow continues where it left off. Remediation is recursive and will delve as deep as necessary to teach the material.

All activities connected directly to INKS nodes by a *statement of objectives*. These identify the competencies or misunderstandings are addressed by this activity. The required list of competencies or misunderstandings is derived from an analysis of the INKS description. Activities can address multiple competencies and misunderstandings. The system will always attempt to select activities that best match the students needs without repeating activities unnecessarily. Activities should also be chosen that adapt to different student learning styles and problem-solving methodologies.

The system will also attempt to exploit different methods of presentation.

2.2 INKS

To master a subject, the student must learn and apply many different types of knowledge. The *Integrated Knowledge System (INKS)* is an organizational methodology that groups and synthesizes these knowledge sources into one common description. This is another core concept used by Siam. It allows the different facets of problem knowledge to be used holistically and gives an educational system considerable flexibility in dealing with the student and the errors they make during learning.

INKS knowledge is represented by a network of information nodes. Each node describes the knowledge necessary to perform one educational task.

Figure 3 presents a pictorially representation of an INKS node. The *node name* identifies this node and is unique from all others.

The *mental model* describes why this node is used.

The *input scene* specifies when this node can be applied to the task. It is a set of preconditions that are expected to be true before the knowledge in this node can be applied.

The *procedure* specifies the set of steps that must be achieved to perform this task.

The *output scene* is the expected result from applying the procedure.

Background concepts are those things implicitly assumed to be known by the student.

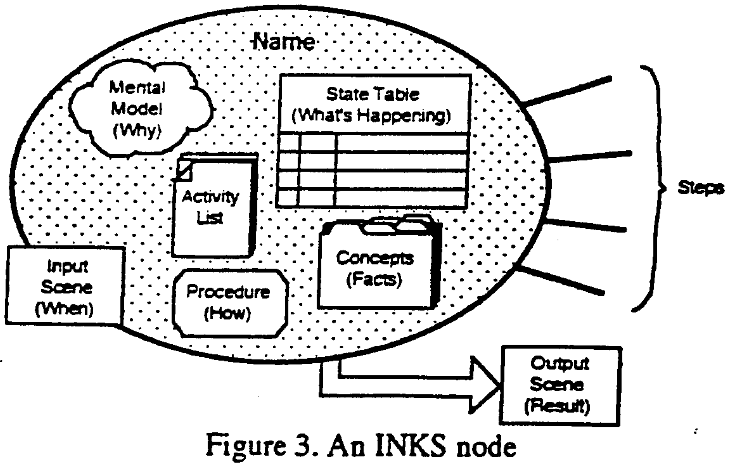


Figure 3. An INKS node

These concepts are declarative in nature and differ from input scenes in their focus. The input scene and mental model describe when and why knowledge should be applied. The background concepts are those things of a more general nature that may be used to reason in the problem domain. Examples include the nature of reptiles or the periodical table.

The *state table* holds information on the student's performance and errors they are making. Each INKS node has one *correct state* (the student know this knowledge), zero or more common error states, and one *unknown error state* (when the student makes an error that is not explicitly represented as a state). The different states are updated as the student is asked questions and responses.

The INKS *activity list* lists all activities that can teach one or more aspects of this node. Each node must have an activity for the mastery standard state and one for each error state. All important concepts have multiple activities which either explain the concept from a different perspective or are designed to address common errors made by students.

In the current system, the mental model, input scene, procedure, output scene, and background concepts are human readable strings and are not interpreted by the software.

The INKS nodes are connected in a directed graph as presented in Figure 4. Each node represents one step in the process that the student must understand. When steps are dependent on other steps, their relationship is represented by a line between the steps. While not shown in Figure 4, an INKS node can have multiple parents.

Subnodes can be placed in an explicit order where all steps must be completed in that order. It is also possible to identify a series of subnodes that form a choice. Only one of these nodes is necessary to complete the task.

2.3 Student Management

Figure 5 is a pictorial representations of the Siam system. Processing begins when the student identifies themselves and selects an allowed curriculum. Siam loads this curriculum's INKS as well as any status the student may have generated on previous uses. If they have not studied this material before, all the INKS nodes are identified as not having been performed.

Most subjects have multiple orders in which the material can be taught. The course map arranges the default order in which activities are taught. This order usually reflects the order of the book being used by the classroom. The course map is actually a activity line and it is loaded with the curriculum.

If the student has already explored this curriculum the system determines if there are any pending remediations to perform and this information if feed to the activity selection.

2.4 Activity Selection

The tutoring system must determine the most appropriate activity to present to the student. The "most appropriate" is influenced by several factors such as the current activity, the state of the student's INKS, prerequisite knowledge, last activity, the phase of the

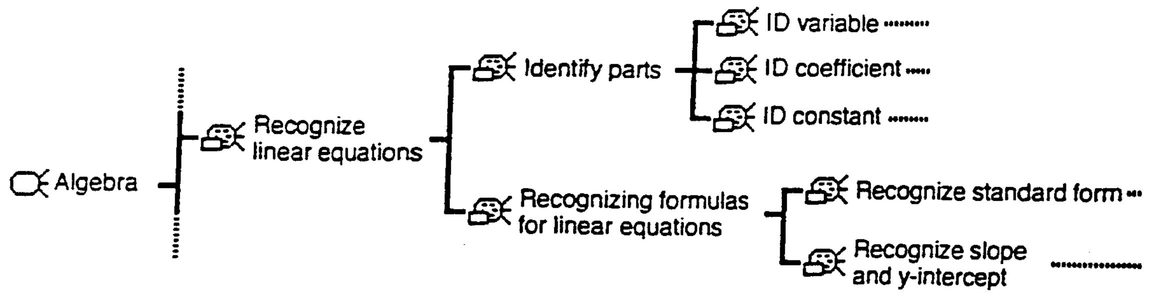
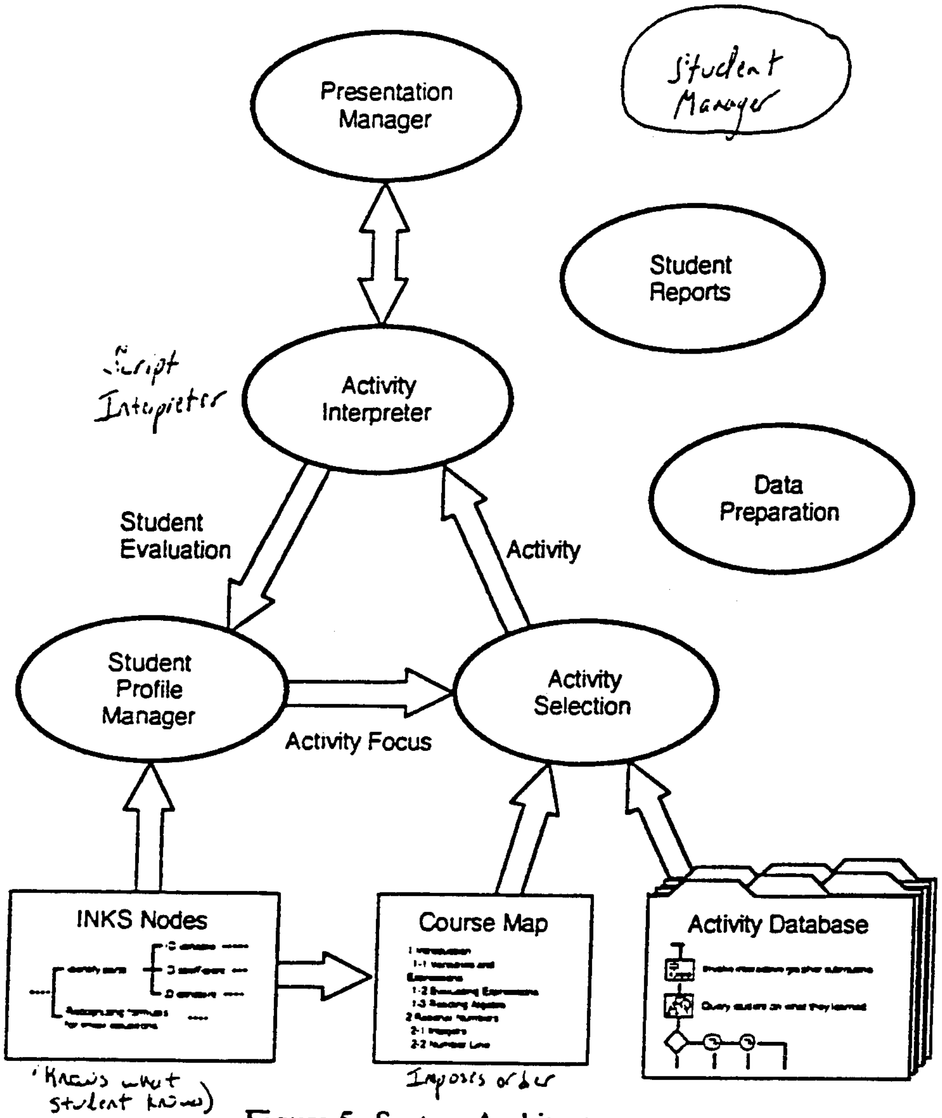

Algebra
Recognize linear equations
Identify parts
ID variable
ID coefficient
ID constant
Recognizing formulas for linear equations
Recognize standard form
Recognize slope and y-intercept

Figure 4. An INKS structure



Remediation is Recursive

Figure 5. System Architecture

moon, and other factors we deem important. The method described in this section is our preliminary approach. It may be modified as we gain experience with the system.

The activity interpreter places evidence against the proper INKS node. At each potential remediation step, these INKS nodes are queried to determine if any remediation is necessary. If none is required, no new activity is chosen and the activity interpreter continues at the next task.

If remediation is required, the activity selector must chose the most appropriate activity. INKS nodes maintain a list of all activities and remediation related to the node. Remediation selection begins by creating a list of all potential activities that can remediate the nodes specified by activity interpreter.

A database keeps a record of all activities shown to the student and an attempt is made to avoid repeating teaching activities.[1] Once repeated activities have been eliminated, the remaining activities have a measure of relevance is calculated for them. This measure is based on the activity's relevance to the current needs of the student. The most relevant activity is chosen and returned to the activity interpreter. The activity interpreter puts the current activity aside and begins the new remediation activity. When the new activity ultimately finishes, the previous activity is continued at its original point in execution.

[1] We may clear this database out when the student restarts. This would allow us to repeat material when enough time had been deemed to elapse.

2.5 Evaluation of Student

In the earlier sections. the actual method to evaluate the student's performance was not discussed. This will now be redressed. As Section 2.2 explains. each INKS node has a list of states representing the student's understanding of the subject. This list has at least two states: the correct state and the unexpected error state. Many will have additional errors that have been identified by the instructional technologist.

As the student works through a activity. they are required to answer questions. The answers to these questions generate evidence that updates the INKS profile of the student's knowledge. The primary objective of the evidence update algorithm is a place the most weight on the recent answers while still remediating previously incorrect answers. As the student continues to answer problems correctly. the INKS node will be declared mastered and the student can move on.

While student answers is the primary method for collecting evidence. another method exists. Attached to many activities are hints for the student. If the students asks for hint either before or after getting and error. negative evidence is added to the INKS node. If. after receiving the hint. they still get the answer wrong. additional evidence is supplied to the node. Once the student has asked for a hint they will not be given credit for this problem, forcing them to demonstrate at a later time that they actually understand the material. This is the algorithm for normal query interactions; games may have a different algorithm.

Appendix A. Glossary

activity - a set of one or more tasks to perform with the student. Tasks are able to present text and graphics. animate graphics. present video or sound. engage in interactive exploration of the information (e.g. an interactive grapher or a game). query for information. assess student responses, and chose future activities.

background concepts - those things of a more general nature that may be used to reason in the problem domain. Examples include the nature of reptiles or the periodical table.

correct state - the state an expert would achieve on the INKS node

INKS - the INtegrated Knowledge System. A fundamental data structure used by our system. It explicitly represents the different aspects of knowledge necessary to understand a field of study.

input scene - specifies when knowledge can be applied to the task. It is a set of preconditions that are expected to be true. It is part of an INKS node.

lessonware - the software description of a curriculum.

mental model - describes why something is being done. It is part of an INKS node.

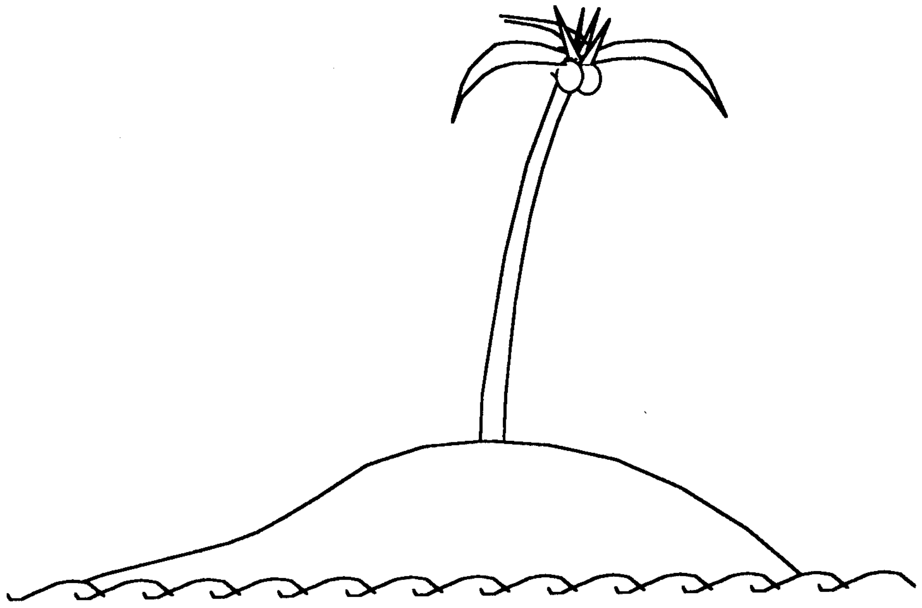
output scene - the expected result from applying the procedure. It is part of an INKS node.

procedure - specifies the set of steps that must be done to perform this task. It is part of an INKS node.

state table - holds information on the student performance and errors they are making

unknown error state - a student error state that has no specific diagnostic information. It is part of an INKS node.

The Gilligan Authoring Language



The ship set ground on the shore this uncharted langugage isle.
With Gilligan, the skipper too, the millionare and his wife, the movie star,
the professor and Mary Anne,
Here on Gilligan's Isle

***G**raphical **I**nteraction **L**anguage for **L**esson **I**ntegration, **G**eneration **A**nd **N**avigation*
RDC Technical Report #95-04

Written by Steve Geyer
Revised and Updated by Jeff Bassett

Introduction

The Gilligan authoring language allows instructional technologists to create, display, and manage computer-based educational materials. The Gilligan language can control both simple and complex behaviours. It language has focused on making simple behaviours easy to describe.

Gilligan lessons can be written in any word processor capable of outputting the document in the Rich Text Format (or RTF).[1] The processing of the RTF file file converts Gilligan commands into educational software.

Need to discuss:

Layers and Areas

Notation for label

Notation for node:state:value

comments and quoting convention.

Commands

All Gilligan commands begin with either an @ or # sign. When @ is used, the command description continues until the end of the paragraph (the first RETURN). '#' is used in only specialized commands and there notation will be described later. Following are a list of Gilligan commands.

Syntax

Throughout the descriptions of the commands, we use the same conventions to describe the syntax. These help to describe what is legal and what is not.

literal	Anything in bold is literal. In other words, that part of the command should appear in the script exactly the same way.
variable	Anything in in italics is variable. It can be replaced by whatever the Instructional Technologist thinks is appropriate or necessary for that command.
[optional]	Anything placed inside of square brackets is optional. It can be part of the command, but doesn't need to be. Most optional items have a default value which is used if the item is omitted.
{repeatable}	This should actually read "repeatable and optional". Any item inside curly brackets can be repeated zero or more times.

[1] While any RTF capable word processor may be used, Gilligan only recognizes the Macintosh RTF equation nototation. Therefore, if equations are part of the lesson, a Macintosh word processor should be used.

@

—

Syntax

@

Function

This is a simple blank line. It is used to separate sections in the Gilligan description without causing text to appear in the instructional material.

Examples

```
@- I want to use a blank line to separate this paragraph from the next
@- one.  Unfortunately that will put a blank line in the lesson also, so
@- I'll use the `@' symbol to make a blank line.
@
@- The `@' symbol can also be used to separate groups of commands.  By
@- separating the groups, it makes it easier for someone else to read
@- and understand what's going on.
```

See also

@-

@-

—

Syntax

@- *some comment string*

Function

All comments begin with @- and continue to the end of the paragraph.

Examples

```
@- The next piece of text will not be displayed until a show command is
used.
@text ▒label1▒ hidden
I'm hidden.  You can't see me.  ha ha.
```

See also

@

@background

—

Syntax

@background *image-name*

image-name refers to a ".bmp" file, so it must be 8 characters or less. It can be either a quoted string or a variable which contains the filename.

Function

Selects a background image.

Examples

```
@background ▒beach▒

@set image ▒beach▒
@background image
```

See also

@graphic
#[picture]

@calculator (not implemented)

Syntax

@calculator *label* [**at** *location*]

label can either be a quoted string or a variable which contains the label name.

Function

Brings up the calculator tool. For now we will be using the calculator that comes with windows.

Examples

```
@text
Try doing a few calculations on the calculator.
@calculator ▒calc1▒
...
erase ▒calc1▒
```

See also

@define

—

Syntax

@define *activity_name* [**remediate** *node:state*[*:value*] { , *node:state*[*:value*] }]

activity_name is always an unquoted string.

Function

In Gilligan, instructional material is constructed from a series of activities. @define is used to create an activity which is made up of a list of commands. All activities must have a label name. They may optionally define a list of INKS node:state:values that are remediated by this activity. In this case, *value* indicates how well this lesson will remediate the given node:state combination.

There is no @end_define command. The activity definition ends when another @define is reached, or the file ends.

Examples

```
@define M1L2S1

@define M3L2R2 remediate node125:state3:1, node15:state1
```

See also

@do

@do

—

Syntax

@do *activity-name*

activity-name can either be a quoted string, or a variable.

Function

@do allows one activity to invoke another activity. When the invoked activity is complete, the original activity continues at the next command.

Examples

```
@background ▒trees▒
@do ▒M1L1I▒
@do ▒M1L1S1▒
@do ▒M1L1P1▒
@do ▒M1L1S2▒
@do ▒M1L1P2▒
@do ▒M1L1S3▒
@do ▒M1L1P3▒
@do ▒M1L1T▒
```

See also

@define

@erase

Syntax

@erase [*object-label*] [**at** *location*]

object-label and *location* can either be quoted strings or a variables.

Function

Erases the object specified. The object can be text, graphics, a numberline or an xy graph. If no object is specified, the entire sector is cleared.

To erase a specific object, that object must have been given a label when it was created.

Examples

```
@- This will erase the entire sector.
@erase

@text ▒text01▒
Better read me fast.
@wait 2
@erase ▒text01▒
```

See also

@text

@evidence (not implemented)

—

Syntax

@evidence *node:state*[*:value*]

Function

Provides evidence to the INKS. *Node* is a short name identifying the node that you have evidence for. *State* is another name identifying either the correct state or one of the error states of that node. *Value* must be a number between 0 and 1 which specifies how strong the evidence is (1 being the strongest). If *value* is ommitted, then the default is 1.

This function is necessary for the numberline and xy graphing tools because neither of them have the capability of automatically updating the INKS.

Examples

```
@numberline ▒n1▒
@text
Plot x > -1.
@numberline_input ▒n1▒ into operation value
@if operation = ▒gt▒ and value = ▒-1▒
   @evidence node25:correctState
@else
   @evidence node25:error1:0.7
@endif
```

See also

@multiple_choice

@graphic

—

Syntax

@graphic *graphic-filename* [**at** *location*]

graphic-filename and *location* can either be a quoted strings or a variables.

Function

Draws the graphic specified by *graphic-filename*. *Graphics-filename* refers to a ".bmp" file, so it should be 8 characters or less. It should also be unique throughout the entire tutor, unlike other labels which only need to be unique within the current activity.

Examples

```
@graphic ▒pic1▒ at ▒top▒
```

See also

#[picture]

@hide

—

Syntax

@hide *label*

label can either be a quoted string or a variable.

Function

Hides an item which is currently displayed, but still leaves space for it.

Examples

```
@text ▒text01▒
This is the first sentence.
@text ▒text02▒
You're not supposed to see this sentence.
@text ▒text03▒
This is the third sentence.
@hide ▒text02▒
```

See also

@show

@text

@highlight

Syntax

@highlight *text-label*

text-label can either be a quoted string or a variable.

Function

Highlights the text specified by *text-label.*

Examples

```
@text ▓sentence1▓
Once upon a time there was this really #{word01,lame} girl named Melinda
Lamowitz.
@- This command highlights the word ▓lame▓ in the above sentence.
@highlight ▓word01▓
@- This command highlights the entire sentence.
@highlight ▓sentence1▓
```

See also

@unhighlight

@if (not implemented)

—

Syntax

@if *expression*
  commands

[@else
  commands]
@end_if

Function

Conditionally executes a set of commands if the expression evaluates to "true". If there is an else clause, then the commands associated with it will be executed if the expression evaluates to false.

The expression is a comparison, usually involving one or more variables. Here is a list of the comparison operators which can be used :

=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal

You can also make complex expressions by combining simple ones with "and", "or" and "not". These operations have an order of precedence: nots are done first, then ands and finally ors. This precedence can be overridden using parentheses.

These expressions can also contain arithmatic. See @set for more details.

Examples

```
@if x = 1
   @erase ▒text01▒
@end_if

@if (y2 - y1) / (x2 - x1) = slope
   @text
   Right.
@else
   @text
   Sorry, that's not the right answer.
@end_if
```

```
@-  Check to see if (x1,y1) equals (0,0) or (1,1)
@if (x1 = 1 and y1 = 1) or (x1 = 0 and y1 = 0)
   Do stuff here.
@end_if
```

See also

@switch

@multiple_choice

Syntax

@multiple_choice [*multiple-choice-label*] [ordered] [at *location*] [into *variable*]

  [rtf-text-of-question]

  { @answer ["*answer-label*"] [correct] [evidence *node:state*[:*value*]]

    rtf-text-of-answer

   [@feedback

     rtf-text-of-feedback]

  }

@end_multiple_choice

Function

This command will ask the student a multiple choice question and wait for a response. The student will be able to select from among several possible answers. These answers are described by using the @answer sub-command. Each @answer can be accompanied by an @feedback command which describes the text that will be displayed if this answer is chosen. Additionally, each @answer can provide evidence to the INKS. The "correct" keyword does not actually indicate anything to the computer. It is used to make the script easier for people to decipher.

By default, the order in which the answers are displayed will be shuffled each time the question is displayed. To override this feature, use the "ordered" keyword.

Examples

```
@multiple_choice
What is your favorite color?
@answer correct
   blue
@feedback
That's right.
@answer
   green
@feedback
No, you're favorite color is blue.
@end_multiple_choice

@- This multiple choice question will update the INKS.
@multiple_choice
Given the equation :  x = 3y + 4
What will x equal if y is 7?
@answer correct evidence node128:state0:1
25
@answer evidence node128:state1:1
7
```

```
@answer evidence node128:state2:1
21
@answer evidence node128:state3:1
11
@end_multiple_choice
```

See also

@evidence

@next

—

Syntax

@next [transition *transition-name*]

Function

@next waits for the user to press the "next" button.

transition-name must be an unquoted string equal to one of the following:

blind
crush
diagonal
drip
explode
random
sand
slide
spiral
split
weave
wipe
snake
slideblind
slideweave
interleave
growlines

Examples

```
@- We want to fade from a page showing ▓pic1▓ to a page showing ▓pic2▓
@graphic ▓pic1▓
@next transition sand
@erase
@graphic ▓pic2▓
@next
```

See also

@define

@numberline (not implemented)

—

Syntax

@numberline [*label*] [start_with *min-value*] [count_by *delta*]

Function

Draws a numberline over the top third of the tutorial area. The numberline will always have 11 tick marks and unless *start_with* and *count_by* are specified it will range from -5 to 5. *Start_with* specifies what the smallest value on the number line will be, and *count_by* specifies the increments between each tick mark.

Up to 2 things can be plotted on any given numberline at one time.

label can either be a quoted string or a variable.

min-value and *delta* can either be numbers or variables containing numbers.

Examples

```
@- Create a numberline which goes from -35 to 15 in increments of 5.
@numberline ▒numline01▒ start_with -35 count_by 5
...
@erase ▒numline01▒
```

See also

@numberline_clear

@numberline_input

@numberline_plot

@numberline_clear (not implemented)

—

Syntax

@numberline_clear *numberline-label*

Function

This function clears any plots that have been made on the numberline so far. It does not erase the numberline itself though. You should use @erase or @erase "label" to do that.

numberline-label can either be a quoted string or a variable.

Examples

```
@numberline ▒nl1▒
@numberline_plot ▒nl1▒ ▒ge▒ -3
@numberline_plot ▒nl1▒ ▒lt▒ 2
@next
@
@- We need to clear the numberline if we want to plot anything more.
@numberline_clear
@numberline_plot ▒nl1▒ ▒eq▒ 4
```

See also

@numberline
@numberline_plot
@numberline_input

@numberline_input (not implemented)

—

Syntax

@numberline_input *numberline-label* **into** *operation value*

after this command, *operation* will equal one of the following:
- **"nothing"** - if the student pressed done without plotting anything.
- **"lt"** - for less than
- **"le"** - for less than or equal to
- **"eq"** - for equal to
- **"gt"** - for greater than
- **"ge"** - for greater than or equal to

numberline-label can either be a quoted string or a variable.

operation and *value* must be variables.

Function

Waits for the student to draw a plot on an existing numberline which was created using the @numberline command. The student will be able to select a point on the numberline and an operation (<, >, <=, >=, =) from a set of buttons drawn under the numberline. Then the student will press a "done" button, and control will return to the script. Information about the plot is returned in two variable: *operation* and *value*. An @if command can then be used to determine whether the student plotted the right thing or not. If the student presses "done" before plotting anything, then *operation* will equal "nothing". It is important to check for this case.

Examples

```
@numberline ▓nl1▓
@---------------- Ask the student to plot something.
@text
Plot the inequality ▓x > 0▓ on the numberline.
@numberline_input ▓nl1▓ into operation value
@
@---------------- Figure out what they plotted.
@if operation = ▓nothing▓
    @text
You didn't plot anything.
@else
    @if operation != ▓gt▓
        @evidence node15:error2:1
        @text
Sorry, you didn't get the right operation.  You should have chosen the
▓greater than▓ or  ▓>▓ button.
    @if value != 0
        @evidence node15:error3:1
        @text
Sorry.  You got the operation right, but you didn't pick the right number
on the numberline.  You should have picked 0.
        @else
```

```
         @evidence node15:correctState:1
         @text
That's right.  Let's go on the the next problem.
      @end_if
   @end_if
@end_if
@
@next
@erase
@----------------- Erase the numberline
@erase ▓nl1▓
```

See also

@if

@numberline

@numberline_plot

@numberline_plot (not implemented)

—

Syntax

@numberline_plot *numberline-label operation value*

where *operation* can be one of the following:
- **"lt"** - for less than
- **"le"** - for less than or equal to
- **"eq"** - for equal to
- **"gt"** - for greater than
- **"ge"** - for greater than or equal to

numberline-label and *operation* can either be a quoted string or a variable.

value can either be a number or a string containing a number.

Function

Draws a plot on an existing numberline (referenced by *numberline-label*).

Examples

```
@- We want to show the student how to plot x < 3
@numberline ▓numline01▓
@numberline_plot ▓numline01▓ ▓lt▓ 3
```

See also

@numberline

@numberline_input

@remediate

Syntax
@remediate

Function
Invokes a remediation cycle. The INKS is examined to see if there are any areas where the student is deficient. If one or more are found, a remediation lesson is found which can address the problem.

Examples
```
@do ▓lesson1▓
@do ▓practice1▓
@do ▓test1▓
@remediate
```

See also
@define

@do

@set (not implemented)

—

Syntax

@set *variable expression*

Function

Sets the *variable* to the value of *expression*. Expressions can be as simple as a number or variable, or as complex as an equation. Here is a list of the arithmatic operators that can be used in expressions:

+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
( )	parentheses are used to change the order of operations

It's important to understand the precedence for the order of operations. What this means is that certain operations are always performed berfore other ones, regardless where they are in the expression. For example, the expressions "3 * x + 5" and "5 + 3 * x" will give the same result because multiplication is always done before addition. Here is the order of precedence, exponentiation is always done first, the multiplication and dividion, and finally addition and subtraction. The parentheses can override this precedence whenever it's necessary. For example, if you're writing the following expression :

$$\frac{y + 5}{x + 1}$$

you must write it as "(y + 5) / (x + 1)". If you don't use the parantheses and you write "y + 5 / x + 1", then what you're really saying is :

$$y + \frac{x}{5} + 1$$

Examples

```
@multiple_choice into y
Pick a number to substitute for `y' in the following equation :
   x = y² + 2y + 3
Then I'll tell you what the answer is.
@answer ▓1▓
1
@answer ▓2▓
2
@answer ▓3▓
3
@answer ▓4▓
```

```
4
@end_multiple_choice
@set x y^2 + 2y + 3
@text
The value of x is #x.
@next
@erase

@text ▓text01▓
Hi there.
@set text_label_var ▓text01▓
@highlight text_label_var
```

See also

@if

@sound (not implemented)

Syntax

@sound *sound-file*

Function

Generates a sound by playing a ".wav" file. The *sound-file* must be 8 characters or less.

sound-file can either be a quoted string or a variable.

Examples

```
@multiple_choice into sound_name
What is the capital of France?
@answer correct ▒chord▒
Paris
@feedback
Right!
@answer ▒blare▒
Berlin
@feedback
Wrong!
@end_multiple_choice
@
@- This will play different sounds depending on whether the answer
@- was right or wrong.
@sound sound_name
```

See also

@graphic

@background

@switch (not implemented)

—

Syntax

@switch *expression*
{ **@case** "*label*"
commands }
[**@default** "*label*"
commands]
@end_switch

Function

Executes the case that has a label matches the the value of *expression*. See @if and for more information on expressions.

Examples

```
@switch x
    @case ▒1▒
            The value of x is 1.
    @case ▒2▒
            The value of x is 2.
    @case ▒3▒
            The value of x is 3.
    @case ▒4▒
            The value of x is 4.
    @default ▒not1-4▒
            The value of x is not 1, 2, 3 or 4.
@end_switch
```

See also

@if

@title (not implemented)

—

Syntax

@title [*label*] [**at** *area-name*]
Some-RTF-text.

Function

@title is used to create a standard title page. All the text following @title will be centered. The first line will be drawn in 24 pnt, the second in 18 pnt and the third in 14 pnt.

label can either be a quoted string or a variable.

Examples

```
@define M1L1S1
@title
Introduction
Module 1
Lesson 1
Section 1
@next
@erase
```

See also

@hide
@show
@highlight
@unhighlight
#{label, text}

@text

—

Syntax

@text [*text-label*] [**at** *area-name*] [**hidden**]
Some-RTF-text.

Function

@text preceeds an rtf text description that will be displayed to the user. The rtf description may include text, equations, and graphic insertions. If a label is included on the @text command line, this is label can be used to control when and how this text is displayed. The **at** attribute specifies where the text will be displayed. If no **at** is specified, the default display area will be used. The **hidden** attribute allows this text to be defined and not displayed on the screen. This hidden text *will* consume its space but it will not be seen (until the author explicitly shows it).

The following text attributes <u>will</u> end up in the text displayed by SIAM.

- Font
- Font size
- Bold, italics, underline
- Margins
- Left and right justification, and centering

text-label can either be a quoted string or a variable.

Examples

```
@text

Here is some text with all sorts of strange_stuff done
                                        to it
to show what will translate into the final lessons.

@text "text01"
You should be able to see this.
@text "text02" hidden
but you won't see me until you press the "next" button.
@
@next
@show "text02"
```

See also

@hide
@show
@highlight
@unhighlight
#{label, text}

@unhighlight

Syntax

@unhighlight *text-label*

text-label can either be a quoted string or a variable.

Function

Highlights the text specified by *text-label.*

Examples

```
@text "sentence1"
Once upon a time there was this really #{word01,lame} girl named Melinda
Lamowitz.
@- This command highlights the word "lame" in the above sentence.
@highlight "word01"
@- This command will unhighlight it.
@unhighlight "word01"
```

See also

@highlight

@xy_graph

Syntax

@xy_graph [*graph-label*] [**start_xy_with** *minx, miny*] [**count_x_by** *xdelta*] [**count_y_by** *ydelta*]

Function

Draws a cartesian coordinate plane in the upper-right quarter of the window. *Start_xy_with* indicates where to start counting on both the x and y axes. For example, if minx and miny are set to 0 and 0, the lower left corner on the graph will be the origin.

Count_x_by and *count_y_by* define the increments between the tick marks on the x and y axes.

graph-label can either be a quoted string or a variable.

minx, miny, xdelta and *ydelta* can either be numbers or variables containing numbers.

Examples

```
@- Draw a graph of the first quadrant.  The tick marks on both
@- the x and y axes will be numbered 0, 10, 20, ... 100.
@xy_graph "graph01" start_xy_with 0,0 count_x_by 10 count_y_by 10
...
@erase "graph01"
```

See also

@xy_graph
@xy_plot_point
@xy_plot_line_2points
@xy_plot_line_yint
@xy_input_point
@xy_input_line_2points
@xy_input_line_yint

@xy_clear

—

Syntax

@xy_clear *graph-label*

Function

Clears away all the plots which have been made on an xy graph. This is mainly for convenience. If you use this command, you won't have to erase and redraw the graph everytime you want to draw something new.

graph-label can either be a quoted string or a variable.

Examples

```
@xy_graph "graph01"
@text
These two line intersect at the origin.
@xy_plot_line_yint "graph01" 1 0
@xy_plot_line_yint "graph01" -1 0
@next
@text
These two lines intersect at the point (2,1).
@xy_clear "graph01"
@xy_plot_line_yint "graph01" 2 -3
@xy_plot_line_yint "graph01" -1 3
@erase "graph01"
```

See also

@xy_graph
@xy_plot_point
@xy_plot_line_2points
@xy_plot_line_yint
@xy_input_point
@xy_input_line_2points
@xy_input_line_yint

@xy_plot_point

—

Syntax

@xy_plot_point *graph-label x, y*

Function

Plots a single point on an xy_graph. Only 2 points can be plotted at any one time.

graph-label can either be a quoted string or a variable.

x and *y* can either be numbers or variables containing numbers.

Examples

```
@- We want to show the student how to plot a line through the
@- point (0,0) and (1,3)
@xy graph "graph01"
@xy plot point "graph01" 1, 3
@xy plot point "graph01" -2, 5
```

See also

@xy_graph

@xy_plot_point

@xy_plot_line_2points

@xy_plot_line_yint

@xy_input_point

@xy_input_line_2points

@xy_input_line_yint

@xy_plot_line_2points

Syntax

@xy_plot_line_2points *graph-label x1, y1 x2, y2* [**hatching** *hatch-style hatch-direction*]

where *hatch-style* can be one of the following:
"line" - just a solid line without hatching (this is the default)
"line_fill" - a solid line with hatching
"dash_fill" - a dashed line with hatching

and *hatch-direction* can be one of the following:
"up" - the hatching is "above" the line - positive y direction
"down" - the hatching is "below the line

If the line is vertical, the "up" direction will fill to the right - the positive x direction.

Function

Draws a plot of the line defined by the two points (x1,y1) and (x2,y2). Can also display an inequality using the "hatching" parameters.

graph-label can either be a quoted string or a variable.

x1, y1, x2 and *y2* can either be numbers or variables containing numbers.

hatch-style and *hatch-direction* can either be a quoted string or a variable.

Examples

```
@- We want to show the student how to plot a line through the
@- point (0,0) and (1,3)
@xy graph "graph01"
@xy plot points "graph01" 0,0 1,3
...
@erase "graph01"
```

See also

@xy_graph
@xy_plot_point
@xy_plot_line_2points
@xy_plot_line_yint
@xy_input_point
@xy_input_line_2points
@xy_input_line_yint

@xy_plot_line_yint

—

Syntax

@xy_plot_line_yint *graph-label slope yintercept* [**hatching** *hatch-style hatch-direction*]

where *hatch-style* can be one of the following:
"line" - just a solid line without hatching (this is the default)
"line_fill" - a solid line with hatching
"dash_fill" - a dashed line with hatching

and *hatch-direction* can be one of the following:
"up" - the hatching is "above" the line - positive y direction
"down" - the hatching is "below the line

Function

Draws a plot of the line defined by the *slope* and *yintercept*. Can also display an inequality using the "hatching" parameters.

graph-label can either be a quoted string or a variable.

slope and *yintercept* can either be numbers or variables containing numbers.

hatch-style and *hatch-direction* can either be a quoted string or a variable.

Examples

```
@- We want to show the student how to plot the inequality
@- y > 3x - 2
@xy_graph "graph01"
@xy_plot_points "graph01" 3 -2 hatching "dash_fill" "up"
...
@erase "graph01"
```

See also

@xy_graph
@xy_plot_point
@xy_plot_line_2points
@xy_plot_line_yint
@xy_input_point
@xy_input_line_2points
@xy_input_line_yint

@xy_input_point

—

Syntax

@xy_input_point *graph-label* **into** *x, y*

Function

Waits for the student to plot a single point on an xy_graph. No more than 2 things can be plotted at any time.

graph-label can either be a quoted string or a variable.

x and *y* must be variables.

Examples

```
@xy_graph "graph01"
@text
Plot the point (1, -2) on the graph.
@xy_input_point "graph01" into x, y
@if x = 1 and y = -2
   Right.
@else
   Wrongo Batman!
@endif
```

See also

@xy_graph
@xy_plot_point
@xy_plot_line_2points
@xy_plot_line_yint
@xy_input_point
@xy_input_line_2points
@xy_input_line_yint

@xy_input_line_2points

—

Syntax

@xy_input_line_2points *graph-label* **into** *x1, y1 x2, y2 hatch-style hatch-direction*

where *hatch-style* can be one of the following:
- **"nothing"** - the student pressed done without completing the plot
- **"line"** - just a solid line without hatching (this is the default)
- **"line_fill"** - a solid line with hatching
- **"dash_fill"** - a dashed line with hatching

and *hatch-direction* can be one of the following:
- **"none"** - either the fill style is "line" or the student pressed done without completing the plot
- **"up"** - the hatching is "above" the line - positive y direction
- **"down"** - the hatching is "below" the line

If the line is vertical, the "up" direction means that the plot is filled to the right - the positive x direction.

Function

Waits for the student to make a plot of a line on the xy graph using two points. If the student presses the "done" button without plotting anything, the value of *hatch-style* will equal "nothing" (this is an important case to check for).

graph-label can either be a quoted string or a variable.
x1, y1, x2, y2, hatch-style and *hatch-direction* must all be variables.

Examples

```
@xy_graph "g1"
@---------------- Ask the student to plot something.
@text
Plot the line which passes through the origin and (2,3)
@xy_input_line_yint "g1" into slope yint hatch_style hatch_direction
@
@---------------- Figure out what they plotted.
@if hatch_style = "nothing"
   @evidence node27:error1:1
   @text
You didn't plot anything.
@else
   @if (x1=0 and y1=0 and x2=2 and y2=3) or (x2=0 and y2=0 and x1=2 and
y1=3)
      @evidence node27:error2:1
      @text
Sorry, you didn't get the right points.
   @else
      @if hatch_style != "line"
         @evidence node27:error4:1
         @text
```

```
Sorry.  The line is in the right place, but it should not be dashed and
there shouldn't be any hatching.
      @else
         @evidence node27:correctState:1
         @text
That's right.  Let's go on the the next problem.
      @end if
   @end if
@end if
@
@next
@erase
@----------------- Erase the xy graph
@erase "g1"
```

See also

@xy_graph
@xy_plot_points
@xy_plot_line_2points
@xy_plot_line_yint
@xy_input_point
@xy_input_line_2points
@xy_input_line_yint

@xy_input_line_yint

—

Syntax

@xy_input_yint *graph-label* **into** *slope yintercept hatch-style hatch-direction*

where *hatch-style* can be one of the following:
- **"nothing"** - the student pressed done without completing the plot
- **"line"** - just a solid line without hatching (this is the default)
- **"line_fill"** - a solid line with hatching
- **"dash_fill"** - a dashed line with hatching

and *hatch-direction* can be one of the following:
- **"none"** - either the fill style is "line" or the student pressed done without completing the plot
- **"up"** - the hatching is "above" the line - positive y direction
- **"down"** - the hatching is "below" the line

Function

Waits for the student to make a plot of a line on the xy graph using the slope and y-intercept. If the student presses the "done" button without plotting anything, the value of *hatch-style* will equal "nothing" (this is an important case to check for).

graph-label can either be a quoted string or a variable.
slope, yintercept, hatch-style and *hatch-direction* must all be variables.

Examples

```
@xy_graph "g1"
@---------------- Ask the student to plot something.
@text
Plot the inequality "y < 2x + 1" on the graph.
@xy_input_line_yint "g1" into slope yint hatch_style hatch_direction
@
@---------------- Figure out what they plotted.
@if hatch_style = "nothing"
    @evidence node27:error1:1
    @text
You didn't plot anything.
@else
    @if slope != 2
        @evidence node27:error2:1
        @text
Sorry, you didn't get the slope right.
    @else
        @if yint != 1
            @evidence node27:error3:1
            @text
Sorry.  You got the slope right, but you didn't intersect the y-axis at the
right place.
```

```
      @else
         @if hatch style != "dash fill"
            @evidence node27:error4:1
            @text
Sorry.  The line is in the right place, but you need to have a dashed line
with hatching below it.
         @else
            @if hatch direction != "down"
               @evidence node27:error5:1
               @text
Sorry.  Everything is right except that the hatching needs to be below the
line instead of above it.
            @else
               @evidence node27:correctState:1
               @text
That's right.  Let's go on the the next problem.
            @end if
         @end if
      @end if
   @end if
@end if
@
@next
@erase
@----------------- Erase the xy graph
@erase "g1"
```

See also

@xy_graph

@xy_plot_point

@xy_plot_line_2points

@xy_plot_line_yint

@xy_input_point

@xy_input_line_2points

@xy_input_line_yint

@wait

Syntax

@wait *time*

Function

Pauses execution for the number of seconds specified by time.

time can either be a number or a variable containing a number.

Examples

```
@wait 3
```

See also

#variable

—

Syntax

#variable

Function

Displays the value of a variable. This can only be used within text.

Examples

```
@text
The value of x is #x.
```

See also

#{label, text}

#[picture]

#[picture]

Syntax
#[*picture*]

Function
Displays a picture at the current location. This can only be used within text.

Examples
```
@text
This lesson consists of 2 sections:
   #[bullet] Adding and subtracting fractions.
   #[bullet] Multiplying and dividing fractions.
```

See also
#variable

#{label, text}

#{label, text}

Syntax

#{*label, RTF-text***}**

Function

Allows labeling of specific words in text.

Examples

```
@text
I want to be able to highlight the word "#{word1,hello}" in this paragraph.
@highlight "word1"
```

See also

#variable

#[picture]

Graphical Interaction Langauge for Lesson Integration, Generation And Navigation (GILLIGAN)

USERS MANUAL

By Deirdre McGlynn

Technical Report #95-03

Research Development Corporation
2875 Towerview Road
Herndon, VA 22071

Scripting with Gilligan

Purpose of this document

The following is a user's guide to the Gilligan scripting language, which allows instructional technologists to create computer based instruction. Scripts should be written in Macintosh RTF.

For the sake of clarity, certain type styles will be used for the example commands of this document. In the example commands of this document, the parts of the commands that appear in the script as written are set in bold face. Specifications (usually in the form of a quoted string) are set in italics. Straight brackets are used to indicate which specifications are optional. The symbol * is used to stand for any string of alphanumeric characters and does not appear in any actual scripts.

@command *"specification"* [*optional-specification*]

In the example scripts of this document (just as in actual scripts), however, only plain text style is used. White spaces never appear within the quote marks in a quoted string. Otherwise, as a general rule, there should be one white space between the command and each of the specifications that modify it. The examples of scripted commands are set in 10 pt font.

```
@text
This is a scripted text item example in ten point font.
```

In the scripts, all commands begin with an @ sign with one type of exception. For commands which are embedded in text, which begin with a # sign. The # sign is also used with variables when they are embedded in text.

Courseware Control

Each course is controlled by the activities that are defined and retrieved. As currently implemented "backgrounds" are controlled at the courseware level. Remediation opportunities are turned on and off as defined by the instructional technologist.

Controlling the sequence of activities

The encoding which handles the sequencing of lessons, practices, and remediations within modules and the organizational structure of the tutor will be done by Drew and Jeff. Jeff will also encode the user menus for the modules and the lesson.

Description of Tutor

The learner is allowed to select the lesson and is given the option of skipping the lesson and proceeding to practice problem sets or tests. There will be in addition an auto-pilot function which chooses an appropriate default path through the tutor for the learner.

Naming convention for activities:

M(1-10)L(1-5)S(1-5)	for each section	M- module
M(1-10)L(1-5)R(1-5)	for each section's remediation	L - lesson
M(1-10)L(1-5)P(1-5)	for each section's practice	T - test
M(1-10)L(1-5)R-L	for each lesson's remediation	R - remediation
M(1-10)L(1-5)P-L	for each lesson's practice	F - filler
M(1-10)L(1-5)T	for each test	
M(1-10)L(1-5)F	for each filler	

Defining activities

Activity is a general term that applies to any section, remediation, filler. At the beginning of each activity file, this command is given to associate the name of the activity with its content.

- to define subsequent script in file as the activity by name (no quotes this time)

  @define *activity*

Ordering activities

The sequence of activities (includes all sections, remediations, fillers) is controlled by the @do command. This command is used at the meta-organizational level rather than in the activity files.

- to retrieve a named activity (use quotes around named activity)

  @do "*activity* "

Remediations

The learner's need for remediation is assessed at the completion of each practice problem set or test. This assessment is based on the learner's performance on the practice and testing sections of the tutor. If indicated by the accumulated evidence, one or more remediations will be selected for the learner at this time. After completing the remediation, students will be given practice problems on the content of the remediation.

- to assess need for remediation; and, if necessary, cues remediation activities for the learner
  @remediate

The script that handles the organizational structure of the tutor will be scripted like this:

```
file "coursmap"

@define COURSEMAP
@do "M1L1"
@do "M1L2"
@do "M1L3"

@do "M2L1"
@do "M2L2"
@do "M2L3"
@do "M2L4"

@do "M3L1"
@do "M3L2"
...
file "M1L1I"

@define M1L1
@background "beach"
@do "M1L1I"
@do "M1L1S1"
@do "M1L1P1"
@remediate
@do "M1L1S2"
@do "M1L1P2"
@remediate
@do "M1L1S3"
@do "M1L1P3"
```

```
@remediate
@do "M1L1P4"
@remediate
@do "M1L1T"
@remediate
```

Backgrounds

A different picture serves as a background for each module and serves as a navigational cue. This pair of commands occurs at the meta-organizational level and does not appear in activity files.

- to place graphic behind tutorial area
  @background *"image_name"*

In this scripted example, a bitmapped graphic of a fern is used as the background for module1.

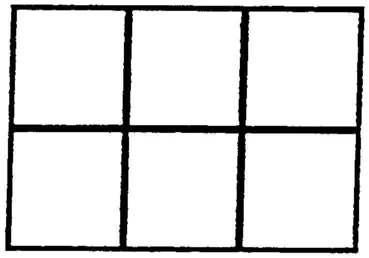
```
@define"mod1fern.bmp"
@background "mod1fern.bmp"
```

Sector Conventions

Sector conventions which specify the location where text/graphic items can be placed are written like this - at "location". The quotes don't include the 'at' and there is one space between the at and quoted word. Don't forget the quotes around the "location"

If a text or graphic item is placed at a specific location, then it appears only within the defined boundaries of that sector.

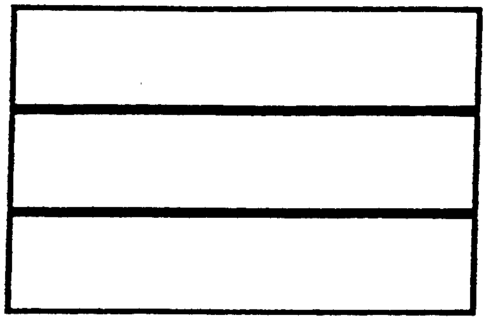
Sector Convention A



this is a scaled model

The tutorial area (blue rectangle) is divided into six sectors. These placements are specified in this way: at "upper, at "lower, plus a _left", -_middle" or _right". (no white spaces are used between these words, e.g. at "lower_left")

Sector Convention B



The tutorial area (blue rectangle) is divided into three sectors -- if not specified otherwise, text wraps into all areas not appropriated by a graphic. These placements are specified in this way: at "top", at "center", and at "bottom".

Miscellany

Comments

All comments begin with @- and continue to the end of the paragraph. Comments themselves never appear on screen, but are used to clarify the meaning of the script.

Here is a scripted example of a comment:

```
@- The next piece of text won't be displayed until later.
@text "text02" hidden
I'm hidden.  You can't see me.  ha ha
```

Skipping lines

The @ sign plus a hard return will be used to leave spaces in the script. This is used only to make the script more legible and does not affect the screen display.

Construction and Control of Instruction

Title pages

Whenever the learner proceeds to a new lesson, section or remediation, a title page will appear. It serves to introduce the learner to the content to come and let the learner know where they are in the tutor. The name of the section will also appear at the top of the screen above the tutorial area and superimposed over the background area.

- to generate a title page-precedes lesson, section, and remediation and always followed by @next/@erase

  `@title`

Example script:

```
@title
This is the title page of this lesson in a generic format
@next
@erase
```

(this command signals a yet-to-be-implemented title page template which is used at the beginning of each lesson, section and remediation)

- to generate a title heading at the top of screen (superimposed over the background)

  `@text at "title"`

<u>Defining the page</u>

The needed paging behaviors are

- progressive reveals
- paging forward
- paging backward

<u>Progressive reveals</u>

Progressive reveals add elements of a series of text and graphic items on the screen one after another as the learner presses the next key. As a rule, erases are not used between items of this series. If one of these items are erased from the screen, items beneath it on the screen will rise higher on the screen to fill the erased area.

- to execute the next command when the learner hits the next key

  `@next`

- to erase the entire screen

  `@erase`

This is a scripted example of a Simple Progressive Reveal:

```
@-the following example was donated by Jeff Bassett and constitutes
evidence that playing Doom promotes violence in the workplace(DM)
@text
Step 1:  Place the rubberband on your right forefinger.
@next
@text
Step 2: Pull the rubberband back with your left hand.
@next
@text
Step 3: Aim at your target (Drew makes a good target).
@next
@text
Step 4: Release the rubberband, and listen to your target scream.
@next
@erase
```

<u>Paging Forward</u>

A page is defined as a "screenful" of text and graphics and delineated by a @next/@erase pair of commands. In essence, the @next/@erase pair of commands acts as a page break.

<u>Paging backward</u>

The learner needs to be able to page backward through a section or lesson or remediation. When the student chooses to hit the back arrow, the previous page starts from the top of the screen, rather than from the last part of a progressive reveal. In practice, the learner is returned to the place in the lesson that corresponds to the immediately previous @next/@erase pair. For the purposes of this function, the @erase must immediately follow the @next and <u>not</u> have any location specified. At present, there is no direct command to override this mechanism, and the hide and show commands will be used to control this process.

Each time the student hits the back key, the program will find the immediately previous @next/@erase pair of commands. Notice there are no comments after the @erase in this instance.

- to create a page break

```
@next
@erase
```

Calling text/graphics to the tutorial area

The tutorial area will have a margin of at least one character all around. If the area within the margin were filled by text, it would contain about16 14 pt lines vertically and about 60 characters horizontally(this information needs to be verified). A graphic item filling the entire available space (including the margin) would be 4.9 in. x 6.9 in.

Placement of text/graphic items

Text and graphic items are placed on the screen according to the current position of the program cursor. (The program cursor is not the same as the mouse/arrow key driven cursor used by the learner. Its location on the screen is controlled by the script and is never seen by the learner.) On an empty page, the default location of program cursor is in the upper left corner of the text area. Therefore, the default placement of the first text/graphic item to be placed on the screen is at the upper left corner as well.

As text or graphics are added to the screen, the program cursor proceeds to the line directly following the last item drawn. In this way, each successive item is placed immediately under whatever item is lowest on the screen, unless a particular sector location is specified.

When an @erase command is executed and the screen is cleared, the program cursor returns to the upper left hand corner again.

About Text

The body of text that follows the @text command is placed on the screen immediately underneath the previous entry unless otherwise specified. RTF will save formatting information on italicizing, underlining, and bolding exactly as the text is entered by the writer of the section or remediation. Font size information will also be saved.

Text that is labeled can also be hidden, which means that the item will consume space, but not appear on screen until the writer explicitly shows it.

- to place a text item on the screen:

```
@text ["label name"][at"center"] [hidden]
```

This example places text on the screen at the current program cursor location:

```
@text
Hi, I'm a piece of text.  What are you?
```

About Graphics

Graphics require filenames that are unique across modules. Graphic filenames are put in quotation marks. These filenames refer to a ".bmp" file, so it should be 8 characters or less. Just like text items, graphics may be labeled. Names of labels need only be unique within a section or remediation and are also within quotes.

- to place a graphic on the screen:
  @graphic "filename" [at *"location"*]

This example places the graphic named coolpic on the screen at the current program cursor location:

```
@graphic "coolpic"
```

This example places the graphic named coolpic on the screen at a specified location:

```
@graphic "coolpic" at "lower_right"
```

This example embeds a picture within text (note that straight brackets are used here):

```
@text
Here is a list of things which are generally a bad idea:
        #[bullet] Playing in the road.
        #[bullet] Running with scissors.
        #[bullet] Kite-flying in thunderstorms
```

Erasing

The erase command can be used to erase the entire screen or a specified area or object. The object can be a text or graphic item, as well as a tool such as a number line or an xy graph. If no object or area is specified, the entire screen is cleared.

- the generic erase command

  @erase["*label-of-item*"] [at"*location*"]

In this example, the entire screen is erased. (Remember that the @next/@erase combination acts like a page break)

```
@- This will erase the entire screen
@erase
```

In this example, the only the text labeled "text01" is erased. (Note that any item underneath "text01" will move up the screen to fill the emptied space if its placement has not been specified)

```
@text “text01”
Better read me fast.
@wait 2
@erase “text01”
```

If a piece of text or graphics is erased from the middle of the screen using @erase “label”, everything which was drawn after it will move up the screen to fill the blank made by the thing which was erased. The program cursor also moves up the screen a corresponding amount.

Labeling

Labels are used to mark text, graphics and the tools. They allow items already placed on the screen to be modified by subsequent commands. Labels are either a quoted strings or a variable. The quoted string can be any alphanumeric beginning with a letter, and must be of eight or fewer characters. No white spaces at all are to be used within the quotation marks of a quoted string.

Labels can either be used for a whole section of text or a whole graphic or for a piece of text or a graphic embedded in text. In relation to text, labels are used for highlighting, hiding, showing and erasing. In labeling blocks of text, place label name in quotes immediately after the @text command.

- To label blocks of text

  @text ["*label-name*"] [at"*location*"] [*hidden*]

In this example of labeled text, notice that no spaces at all are used between the quotes:

```
@text "text01" at "middle" hidden
If this were unhidden text, you would be able to read this.
```

For the purpose of marking specific pieces of text within a block of text for highlighting, hiding, showing and erasing inline labels are used.

- to label as * the piece of text or graphic item within curly-brackets
  #{ * , *item labeled* }

To label a specific word within a body of text, script it this way:

```
@text
There once was this really #{word01,brilliant} girl named Melinda
Brillowitz.
```

Hiding/Showing

Once a text or graphic item is given a label, the writer can hide/show the item by using the appropriate command followed by its label. Unlike the screen behavior when an item is erased, when an item is hidden, the cursor does not move back up the screen. Therefore, an item whose location is not specified and which is lower down on the screen will move up to fill the erased area unless the removed item is "hidden" rather than erased. Hidden text will consume space but it will not be seen until the author explicitly gives the command to show it.

- to hide a labeled text or graphic item
  @hide *"label01"*

- to show a labeled text or graphic item
  @show *"label01"*

This is a scripted example of hide/show behavior in a progressive reveal in which text is revealed in the middle of the screen

```
@text
Step 1
```

```
@—- This text starts out hidden and will be revealed later.
@text "text02" hidden
Step 2
@text
Step 3
@next
@show "text02"
@next
@erase
```

This is the page as it first appears:

```
Step 1

Step 3
```

This is how the page looks after pressing "next":

```
Step 1
Step 2
Step 3
```

This is another example of a hide/show script in a progressive reveal in which text items are replaced on the screen as you progress.

```
@text "text01"
Step 1
@text "text02a"
Step 2 (form a)
@text "text03"
Step 3
@next
@
@erase "text02a"
@—– The next erase acts as a "cut"
@erase "text03"
@text "text02b"
Step 2 (form b)
@—– This show acts as a "paste"
@show "text03"
@next
@
@erase "text02b"
@—– The next erase acts as a "cut"
@erase "text03"
@text "text02c"
Step 2 (form c)
@—– This show acts as a "paste"
@show "text03"
```

```
@next
@erase
```

The page as it first appears:

```
Step 1
Step 2 (form a)
Step 3
```

This is how the page looks after pressing "next":

```
Step 1
Step 2 (form b)
Step 3
```

This is how the page looks after pressing "next" a second time:

```
Step 1
Step 2 (form c)
Step 3
```

<u>Highlighting</u>

Once a text is given a label, the writer can highlight/unhighlight the item by using the appropriate command followed by its label.

- to highlight a labeled text
  `@highlight` *"label01"*

- to unhighlight a labeled text
  `@unhighlight` *"label01"*

An scripted example of highlight/unhighlight behavior:

```
@text "text01"
Pay attention to this text.
@highlight "text01"
```

A scripted example of highlight/unhighlight behavior for a specific word within a body of text:

```
@text
There once was this really #{word01, brilliant} girl named Melinda
Brillowitz.
@next
@highlight "word01"
```

The word “brilliant” is highlighted in the above text when the learner presses next.

Transitions/Sounds/Waits

Transitions

Transitions will be used to emphasize a logical progression in a series of graphics.

- to use an effect
  @next [transition *transition_name*]

Available transition effects are as follows:
(*transition-name* must be an <u>unquoted</u> string):

blind	crush
diagonal	drip
explode	random (approximates fade)
sand	slide
spiral	split
weave	wipe
snake	slideblind
slideweave	interleave
growlines	

This is a scripted example of a transition effect - note that the word random is unquoted in this instance:

```
@- We want to fade from a page showing “pic1” to a page showing
“pic2”
@graphic “pic1”
@next transition random
@erase
@graphic “pic2”
@next
```

Sound Cues

Three types of sounds will be used to emphasize points or give feedback to the learner. Chord indicates a correct response, blare, an incorrect response and ring is used for emphasis. As with labels, sound files are a quoted string of eight or fewer characters.

- to play a given sound
  @sound *#"sound_file"*

Here is a scripted example of the use of a sound cue:

```
@sound #"chord"
```

Wait

A wait command can also be used to delay entering additional items to the screen in the absence of the learner pressing the next key. Only time increments of whole seconds are permitted.

- to delay execution of the next command for a number of seconds
  @wait *number_of_seconds*

A scripted example of wait behavior lasting 3 seconds:

```
@wait 3
```

Gathering INKS Node Evidence

In this tutor, the evidence sent to an INKS node is generated only from practice problems and from test questions (locally, this process is known as INKSifying). However, evidence can be gathered from any point is the tutor.

A multiple choice question from which evidence is gathered will have the following notation after the @answer command. This set is reiterated for each node that is referenced:

- to INKSify a multiple choice question
  @multiple_choice [evidence *node:state* [*:value*]]

(See scripted example in multiple choice section below)

Any other decision points from which the writer may wish to gather evidence will use an @evidence command. This decision point may be either an if statements or switch command. The @evidence command is reiterated for each node that is referenced:

- to INKSify an if statement or switch command

  @evidence *node:state* [*:value*]

  (See scripted example in section on if statements and switches below)

At the @remediate command, the accumulated evidence is weighed, and if it is determined that a remediation is needed, the appropriate remediation section is sent to the learner.

Multiple Choice Questions

The order of the responses to the multiple choice questions are automatically randomized unless writer specifies that responses are to be presented in the order they have been scripted.

Every multiple choice question must contain all of the following:

- to display mc question to the screen (order of answers is randomized unless ordered is specified)

  @multiple_choice [*ordered*]

- to display each answer to the screen

  @answer [*correct*] [evidence *node:state* [*:value*]]

(Each @answer can provide evidence to the INKS. The "correct" keyword or lack of it does not actually indicate anything to the computer. It is used to make the script easier for people to decipher and edit.)

- to execute display of text only if the immediately previous @answer has been selected by the learner

  @feedback

- to signal end of the mc question

  @end_multiple_choice

This is a scripted example of a multiple choice item from which evidence is not collected:

```
@multiple_choice
Which of the equations below describes the word problem?
@answer
4.5 - x = 7.2
@feedback
That's right
@answer
4.5 = 7.2 + x
@feedback
Even more than that.
@answer correct
4.5 + x = 7.2
@feedback
At the very least.
@end_multiple_choice
```

This is a scripted example of an INKSified multiple choice item :

```
@- This multiple choice question will update the INKS.
@multiple_choice
Given the equation : x = 3y + 4
What will x equal if y is 7?
@answer correct evidence node128:state0:1
25
@answer evidence node128:state1:1
7
@answer evidence node128:state2:1
21
@answer evidence node128:state3:1
11
@end_multiple_choice
```

Variables

Variables are used as a container for a given value. This command sets a variable to the value of a selected expression. Expressions can be as simple as a number or text item, or as complex as an equation.

- to set a variable to the value of an expression
  @set *variable expression*

- to expand a variable (can be text or graphic item) within a text block:

```
@text
_________________________
____#var ____________
-------------------------
```

Here's an example of setting a variable to a text item:

```
@set greeting "Hello"
@text
This little piece of text says #greeting
@-the preceding @text displays as – This little piece
of text says "Hello"
```

Putting in numbers works in the same way. If the variable in the case above were set to 10, the number 10 will be expanded in the text in the place of the #var.

Variables can also be defined according to learner response to a multiple choice question. The writer can thus use variables to personalize the instruction for the learner.

- to define a variable through learner response to a multiple choice question

  @multiple_choice [into *var*]

For example, the learner can be allowed to choose his own variable for a subsequent problem. Notice that the <u>label</u> is what is put into the variable, not the answer itself. Here is a scripted example:

```
@multiple_choice into var1
Pick any letter to stand for the unknown in the problem.
@answer "x"
 x
@answer "y"
 y
@answer "z"
 z
@end_multiple_choice
```

If the student has chosen the first answer, the variable is given a value of 'x'.

Here is an example in which the learner chooses the value with which to evaluate an equation.

```
@multiple_choice into y
Pick a number to substitute for "y" in the following equation :
x=y^2 + 2y+3
then I'll tell you what the answer is.
@answer "1"
1
@answer "2"
2
@answer "3"
3
@answer "4"
4
@end_multiple_choice
@set x y^2 + 2y + 3
@text
The value of x is #x.
@next
@erase
```

A variable could also be used to provide audio feedback in multiple choice problems. Here is a scripted example of audio feedback:

```
@multiple_choice into sound_file
What the right answer?
@answer #"blare"
a wrong answer
@answer #"blare"
another wrong answer
@answer #"chord"
The right answer
@end_multiple_choice
@sound sound_file
```

More about expressions

Here is a list of the arithmetic operators that can be used in expressions in the order that they are performed:

( )	operations within parentheses
^	exponentiation
• and /	multiplication and division
+ and -	addition and subtraction

Branching according to Learner Interactions

At times, the sequence of activities or feedback presented to the learner may be adjusted according to previous learner interactions. An if statement or a switch is used to define the conditions under which this branching takes place. If statements will be used to evaluate learner interactions with the tools, and give appropriate feedback to the learner. The learner's responses to an if statement or a switch can also furnish evidence for an INKS node.

If statements

The if statement is generally used rather that the switch command when there are only two alternatives. Additionally, each @if or @else can provide evidence to the INKS when followed by an @evidence command.

Every if statement contains all of the following components:

- to begin an if statement:
  @if *expression*
  (The commands listed after @if will be executed, if the condition indicated by the expression is true)

- to specify feedback or activity sequence if the expression is not true (this is optional in an if statement)
  @else
  (the commands listed after @else will be executed, if the condition indicated by the expression after @if is not true)

- to mark the end of an if statement (this is *not* optional):
  @end_if

If the condition indicated by the expression is not true and there is no @else command, then the entire script between @if and @end_if is ignored.

Here is a scripted example of an if statement:

```
What is does #var1^var1 + 1 equal?
@answer "8"
      8
@answer "81"
      81
@answer "1024"
      1024
@end_multiple_choice
@
@if var2 = var1 ^ (var1 + 1)
That's right!
@else
Sorry that's wrong.
@end_if
```

Here is a list of the comparison operators which can be used in the *expression* clause:

=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal

You can combine these simple comparison operators to make complex expressions with "and", "or" and "not".

A scripted example of a complex expression:

```
@if var = 1 or var >= 3
@highlight "word01"
@end_if
```

Switch

Switches can be used to synthesize several if statements when there are several alternatives. Each case listed after a @switch corresponds to a single if statement. The case that has the label which matches the value of the variable is the one that will be executed. Additionally, each @case can provide evidence to the INKS when followed by an @evidence command.

Every switch contains all of the following components:

- to begin a switch (the variable may be set to an expression)
  @switch *variable*

Following this a list of alternative cases that are labeled. The case in which the label matches the variable of the @switch is executed.

- to list each alternative case (there will be several of these)
  @case *"label"*
  (plus the list of commands to be executed if case label matches the variable)

- selected if none of the cases apply
  @default *"label"*
  (plus the list of commands to be executed)

- to end a switch
  @end_switch

This is a scripted example of a switch:

```
@switch x
        @case "1"
                The value of x is 1.
        @case "2"
                The value of x is 2.
        @case "3"
                The value of x is 3.
        @case "4"
                The value of x is 4.
        @default "not1-4"
                The value of x is not 1, 2, 3 or 4.
@end_switch
```

Tools

All tools called to the screen by the writer of instruction must be labeled. Variables can be used to record and evaluate the learner's interactions with the graphing tools. If a graphing tool is called by the writer, a command can be used to enter coordinate points or lines which the writer wishes to be displayed upon opening the tool. All tools must be explicitly erased by the label given to them by the writer rather than with the @erase tool.

- Number line
- Xy graph
- Calculator
- Notepad

Number line

The number line will always have 11 tick marks and unless start_with and count_by are specified it will range from -5 to 5. Start_with specifies what the smallest value on the number line will be, and count_by specifies the increments between each tick mark. If the writer wishes to give the learner the option of calling the number line, this choice can be offered through a multiple choice question

The placement of the number line is at top of screen and centered (may overlay part of background layer, but not title at top of screen) The window in which the number line appears is not a movable window.

- to call a number line

  @numberline *"numberline-label"* [start_with *min-value*]
  [count_by *delta*]

- to erase number line

  @erase *"numberline-label"*

An example of calling the number line to the screen and erasing it.

```
@- Create a number line which goes from -35 to 15 in increments of 5.
@number_line "numline01" start_with -35 count_by 5
@erase "numline01"
```

Operations with number lines

These abbreviations will indicate the noted operation:

nothing - if the student pressed done without plotting anything.
lt - for less than
le - for less than or equal to
eq - for equal to
gt - for greater than
ge - for greater than or equal to

These operations can be used to form one variable expressions which can be plotted on the numberline.

A total of 2 points or simple expressions can be plotted on any given number line at one time. The first item plotted is displayed in blue and the second is in green, regardless of whether it has been plotted by the writer or learner.

In order to enter two plots on the number line, the writer uses the @numberline_plot twice.

- to plot a point or an expression on a number line
  @numberline_plot *numberline-label operation value*

```
@-we want to show the learner how to plot x > 3
@numberline "numline01"
@numberline_plot "numline01" "gt" 3
```

When the learner is asked to plot a point on the number line and the command below is given, a set of buttons representing operations (<, >, <=, >=, and =) will appear to allow the learner to control the plotting. The learner will be able to select a point on the number line and one of the operations. If the learner is asked to plot two points in a number line, a second @numberline_input command controls the second input.

- to indicate the learner will select a point or a point and an operation and to return input data from the learner interaction's with the number line for evaluation.

  @numberline_input "*numberline-label*" into *operation value*

An @if/@else script can be used to evaluate the correctness of the student's answer and give the appropriate feedback to the learner. The following is an example of using an if statement to give feedback.

```
@numberline "nl1"
@-------------- Ask the student to plot something.
@text
Plot the inequality "x > 0" on the numberline.
@numberline_input "nl1" into operation value
@
@-------------- Figure out what they plotted.
@if operation = "nothing"
   @text
   You didn't plot anything.
   @else
      @if operation != "gt"
      @evidence node15:error2:1
      @text
      Sorry, you didn't get the right operation. You should have chosen the
      "greater than" or ">" button.
            @if value != 0
            @evidence node15:error3:1
            @text
            Sorry. You got the operation right, but you didn't pick the
            right number on the numberline. You should have picked 0.
               @else
               @evidence node15:correctState:1
               @text
               That's right. Let's go on the next problem.
      @end_if
   @end_if
@end_if
@next
@erase
@---------------- Erase the numberline
@erase "nl1"
```

Only the latest plot made on the number line so far can be cleared from the screen by the learner control. It does not erase the number line itself though. You should use @erase "label" to do that. The @erase will not remove the numberline.

- to clear any plots that have been made to the number line
  @numberline_clear *"numberline-label"*

A scripted example of clearing the numberline:

```
@numberline "nl1"
@numberline_plot "nl1" ge -3
@numberline_plot "nl1" lt 2
@next
@- We need to clear the number line if we want to plot anything
more.
@numberline_clear
@numberline_plot "nl1" eq 4
```

Xy graph

The placement of the xy graph is at far upper right (overlays part of background layer). The window in which the xy graph appears is not a movable window. There is a point plotting mode and a line plotting mode available. This modality is determined by the subsequent command.

Each axis of the xy graph will always have 11 tick marks and unless start_with and count_by are specified, both axes will range from -5 to 5, counting by ones. Start_xy_with specifies what the smallest value on each axis will be, and count_by specifies the increments between each tick mark

As with the number line, the first item plotted is displayed in blue and the second is in green (either 2 points or 2 lines). Whether it is the writer or learner who makes the plot makes no difference in this regard.

Only the writer of instruction can call the xy graph to the screen. However, if the writer wishes to give the learner the option of calling the xy graph, this choice can be offered through a multiple choice question.

- to call the xy graph
  @xy_graph *"xy_label"* [start_xy_with *minx* ,*miny*] [count_x_by*xdelta*] [count_y_by*ydelta*]

- to erase the xy graph
  @erase *"xy_label"*

Here is an example of a scripted call and erasure of an xy graph. Note that each axis begins with zero and goes up to 100, counting by tens:

```
@xy_graph "graph01" start_xy_with 0,0 count_x_by 10 count_y_by
10
@erase "graph01"
```

Plotting points on an xy graph

A total of two coordinate points per graph can be either plotted by the writer or inputted by the student. The x, y used below in the may either be numbers or be variables containing numbers.

- to plot a single point on an xy_graph.
  @xy_plot_point *"graph_label" x, y*

In this example, the @xy_plot_point command is repeated to add a second point to the xy graph

```
@- We want to show the student how to plot the points (0,0) and (1,3)
@xy_graph "graph01"
@xy_plot_point "graph01" 0, 0
@xy_plot_point "graph01" 1, 3
```

- to indicate the learner will enter a single point on an xy_graph and to set the variables x and y according to the learner interaction with the xy graph.
  @xy_input_point *"graph-label"* into *x* , *y*

Here is an example script:

```
@xy_graph "graph01"
@text
Plot the point (1, -2) on the graph.
@xy_input_point "graph01" into x, y
@if x = 1 and y = -2
Right.
```

```
@else
Wrongo Batman!
@end_if
```

Plotting lines on the xy graph

A total of two lines per graph can be either plotted by the writer or inputted by the student. The first line drawn appears in blue and the second line drawn is in green.

There are two ways to plot or input lines on the xy graphs. The first plots two points and draws a line between them. The second plots the y-intercept and slope. As with the number line, learners can revise their plots by using the erase button attached to the graph. After they press done, though, the program moves on to the next question.

Hatching is also employed to display inequalities. Hatching is specified by direction and style.

- hatch-style can be one of the following:
  line - just a solid line without hatching (this is the default)
  line_fill - a solid line with hatching
  dash_fill - a dashed line with hatching

- hatch-direction can be one of the following:
  up - the hatching is "above" the line - positive y direction
  down - the hatching is "below the line

If the line is vertical, the "up" direction will fill to the right (the positive x direction) and the "down" direction will fill to the left (the negative x direction.

The x, y used in the command below in the may either be numbers or be variables containing numbers. Hatch-style and hatch-direction will also be either a quoted string or a variable. One white space is used between the hatch-style and hatch-direction in the following command as well as between each pair of xy coordinates.

- to plot a line on an existing graph (referenced by *graph-label*) by specifying two pairs of xy coordinate.

  @xy_plot_line_2points *"graph-label" x1, y1 x2, y2* [hatching *hatch-style hatch-direction*]

An example of a two coordinate-type line plot script:

```
@- We want to show the student how to plot a line through the
@- point (0,0) and (1,3) with hatching above
@xy_graph "graph01"
@xy_plot_line_2points "graph01" 0,0 1,3  hatching"line_fill" "up"
```

- to a line plot on an existing graph (referenced by *graph-label*) by specifying the slope and the y intercept.

  @xy_plot_line_yint *graph-label slope yintercept* [hatching *hatch-style hatch-direction*]

An example of a two coordinate-type line plot

```
@- We want to show the student how to plot y > 3x - 2
@xy_graph "graph01"
@xy_plot_line_yint "graph01" 3  -2 hatching "dash_fill" "up"
```

Clearing the xy graph

The graph can be cleared of anything plotted on it in order to prepare it for the next data entries. This is an easier alternative to erasing it and redrawing the graph from scratch.

- to clear previous plots or inputs on the graph

  @xy_graph_clear

This is an example of clearing plots on the xy graph:

```
@xy_graph "graph01"
@text
These two line intersect at the origin.
@xy_plot_line_yint "graph01" 1 0
@xy_plot_line_yint "graph01" -1 0
@next
@text
These two lines intersect at the point (2,1).
@xy_clear "graph01"
@xy_plot_line_yint "graph01" 2 -3
@xy_plot_line_yint "graph01" -1 3
@erase "graph01"
```

Calculator

The calculator can be called to the screen or erased on demand by both the learner and the writer of the instruction. The learner does this through the pop-up window brought up by the Display Tools icon. The window in which the calculator appears is a movable window and its placement can be adjusted by the learner. The writer uses the following commands to do the same thing.

- to call calculator

  @calculator *"label"* [at *"location"*]

A specific erase command must be used to erase the calculator as the regular @erase will not affect it.

- to erase calculator

  @erase *"label"*

Here is a scripted example:

```
@text
Try doing a few calculations on the calculator.
@calculator "calc1"
erase "calc1"
```

Notepad

Default placement for this tool is currently undefined. Default placement may be defined as placement at top of screen and centered (may overlay part of background layer, but not title at top of screen) There is no scripting call for this function.

Help

The help function will include a navigation guide and the glossary.

Glossary

The glossary is accessed though the help function. Default placement for this function is currently undefined. Default placement may be defined as placement at far upper right (overlays part of background layer) It is called to the screen and erased by the glossary icon. Learner can search for glossary words by alphabetical index. There is no scripting call for this function.